The Application Kit

Framework: Library/Frameworks/AppKit.framework

Header File Directories: Library/Frameworks/AppKit.framework/Headers

Introduction

The Application Kit is a framework containing all the objects you need to implement your graphical, event-driven user interface: windows, panels, buttons, menus, scrollers, and text fields. The Application Kit handles all the details for you as it efficiently draws on the screen, communicates with hardware devices and screen buffers, clears areas of the screen before drawing, and clips views. The number of classes in the Application Kit may seem daunting at first. However, most Application Kit classes are support classes that you use indirectly. You also have the choice at which level you use the Application Kit:

- Use Interface Builder to create connections from user interface objects to your application objects. In this case, all you need to do is implement your application classes—implement those action and delegate methods. For example, implement the method that is invoked when the user selects a menu item.
- Control the user interface programmatically which requires more familiarity with Application Kit classes and protocols. For example, allowing the user to drag an icon from one window to another requires some programming and familiarity with the NSDragging... protocols.
- Implement your own objects by subclassing NSView or other classes. When subclassing NSView you write your own drawing methods using graphics functions. Subclassing requires a deeper understanding of how the Application Kit works.

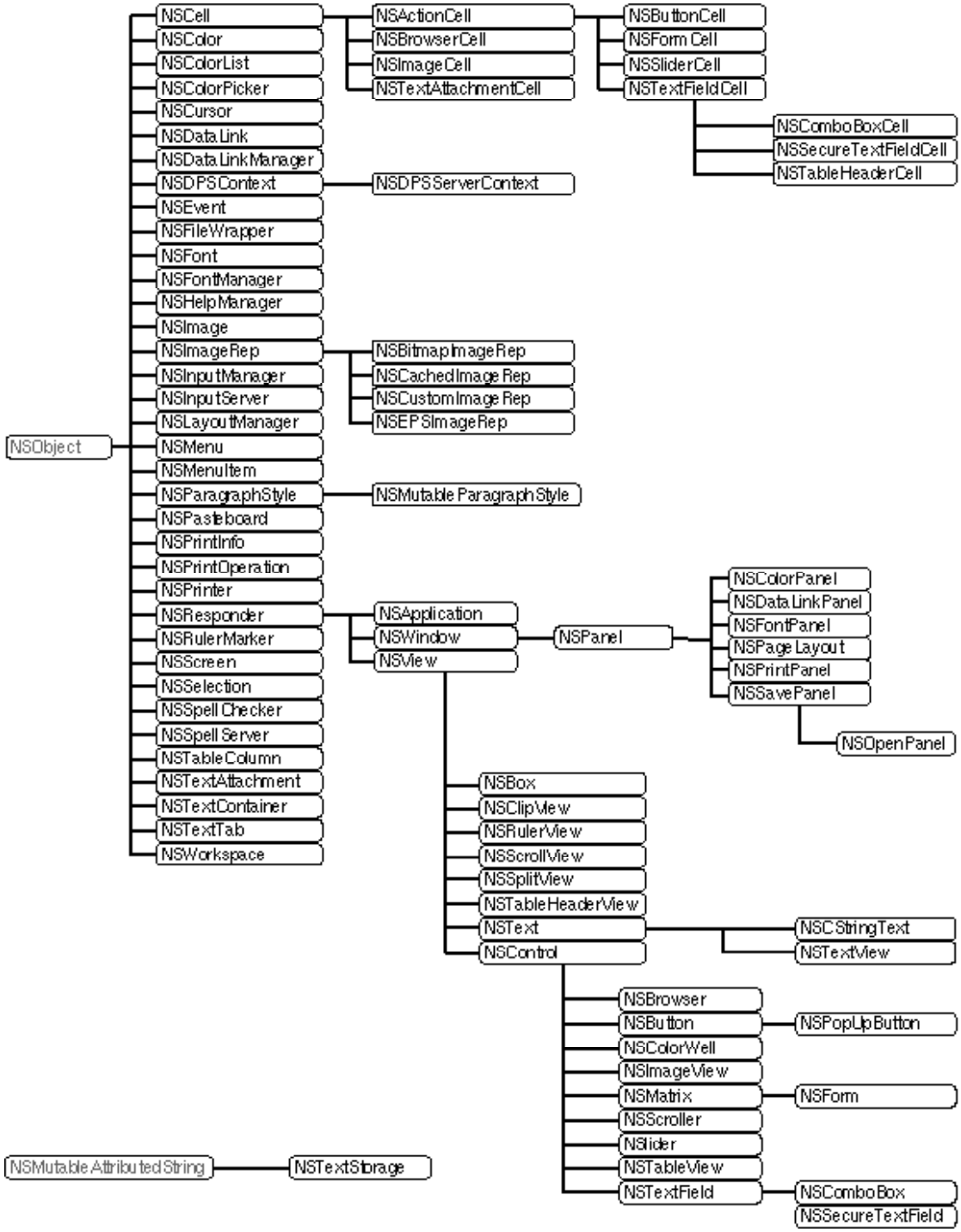
To learn more about the Application Kit, review the NSApplication, NSWindow, and NSView class specifications paying close attention to delegate methods. For a deeper understanding of how the

Application Kit works, see the specifications for NSResponder and NSRunLoop (NSRunLoop is in the Foundation Framework).

Application Kit Classes and Protocols

The Application Kit is large; it comprises more than 100 classes and protocols. The classes all descend from the Foundation Framework's NSObject class (see Figure 1). The following sections briefly describe some of the topics that the Application Kit addresses through its classes and protocols.

Figure 1 The Application Kit class inheritance



Encapsulating an Application

Every application uses a single instance of NSApplication to control the main event loop, keep track of the application's windows and menus, distribute events to the appropriate objects (that is, itself or one of its windows), setup autorelease pools, and receive notification of application-level events. An NSApplication object has a delegate (an object that you assign) that is notified when the application starts or terminates, is hidden or activated, should open a file selected by the user, and so forth. By setting the NSApplication object's delegate and implementing the delegate methods, you customize the behavior of your application without having to subclass NSApplication.

General Event Handling and Drawing

The NSResponder class defines the responder chain, an ordered list of objects that respond to user events. When the user clicks the mouse or presses a key, an event is generated and passed up the responder chain in search of an object that can "respond" to it. Any object that handles events must inherit from the NSResponder class. The core Application Kit classes, NSApplication, NSWindow, and NSView, inherit from NSResponder.

An NSApplication object maintains a list of NSWindow objects—one for each window belonging to the application—and each NSWindow object maintains a hierarchy of NSView objects. The view hierarchy is used for drawing and handling events within a window. NSWindow objects handle window-level events, distribute other events to its views, and provide a drawing area for its views. An NSWindow object also has a delegate allowing you to customize its behavior.

NSView is an abstract class for all objects displayed in a window. All subclasses implement a drawing method using graphics functions; this is the primary method you override when creating a new NSView.

Panels

The NSPanel class is a subclass of NSWindow that you use to display transient, global, or pressing information. For example, you would use an instance of NSPanel, rather than an instance of NSWindow, to display error messages, or to query the user for a response to remarkable or unusual circumstances. The Application Kit implements some common panels for you such as the Save, Open and Print panels, used to save, open, and print documents. Using these panels gives the user a consistent "look and feel" across applications for common operations.

Menus and Cursors

The NSMenu, NSMenuItem, and NSCursor classes define the look and behavior of the menus and cursors that your application displays to the user.

Grouping and Scrolling Views

The NSBox, NSScrollView, and NSSplitView classes provide graphic "accessories" to other view objects or collection of views in windows. With the NSBox class, you can group elements in windows and draw a

border around the entire group. The NSSplitView class lets you “stack” views vertically or horizontally, apportioning to each view some amount of a common territory; a sliding control bar lets the user redistribute the territory among views. The NSScrollView class, and its helper class, NSClipView, provide a scrolling mechanism as well as the graphic objects that let the user initiate and control a scroll. The NSRulerView class allows you to add a ruler and markers to a scrollview.

Controlling an Application

The NSControl and NSCell classes, and their subclasses, define a common set of user interface objects such as buttons, sliders, and browsers that the user can manipulate graphically to control some aspect of your application. Just what a particular control affects is up to you: When a control is “touched,” it sends an action message to a target object. You typically use Interface Builder to set these targets and actions by control-dragging from the control object to your application or other object. You can also set targets and actions programmatically.

An NSControl object is associated with one or more NSCell objects that implement the details of drawing and handling events. For example, a button comprises both an NSButton object and an NSButtonCell object. The reason for this separation of functionality is primarily to allow NSCell classes to be reused by NSControl classes. For example, NSMatrix and NSTableView can contain multiple NSCell objects of different types.

Tables

The NSTableView class displays data in row and column form. NSTableView is ideal, but not limited to, displaying database records, where rows correspond to each record and columns contain record attributes. The user can edit individual cells and rearrange the columns. You control the behavior and content of an NSTableView object by setting its delegate and data source objects.

Text and Fonts

The NSTextField class implements a simple editable text field, and the NSTextView class provides more comprehensive editing features for larger text bodies.

NSTextView, a subclass of the abstract NSText class, defines the interface to OpenStep’s extended text system. (Use only the methods defined by NSText if you are programming strictly according to the OpenStep Specification.) NSTextView supports rich text, attachments (graphics, file, and other), input management and key binding, and marked text attributes. NSTextView works with the font panel and menu, rulers and paragraph styles, the Services facility (for example, the spell-checking service), and the pasteboard. NSTextView also allows customizing through delegation and notifications—you rarely need to subclass NSTextView. You rarely create instances of NSTextView programmatically either since objects on Interface Builder’s palettes, such as NSTextField, NSForm and NSScrollView, already contain NSTextView objects.

It is also possible to do more powerful and more creative text manipulation (such as displaying text in a circle) using NSTextStorage, NSLayoutManager, NSTextContainer, and related classes.

The NSFont and NSFontManager classes encapsulate and manage font families, sizes, and variations. The NSFont class defines a single object for each distinct font; for efficiency, these objects, which can be rather large, are shared by all the objects in your application. The NSFontPanel class defines the font-specification panel that's presented to the user.

Graphics and Color

The classes NSImage and NSImageRep encapsulate graphic data, allowing you to easily and efficiently access images stored in files on the disk and displayed on the screen. NSImageRep subclasses each know how to draw an image from a particular kind of source data. The presentation of an image is greatly influenced by the hardware that it's displayed on. For example, a particular image may look good on a color monitor, but may be too "rich" for monochrome. Through the image classes, you can group representations of the same image, where each representation fits a specific type of display device—the decision of which representation to use can be left to the NSImage class itself.

Color is supported by the classes NSColor, NSColorPanel, NSColorList, NSColorPicker, and NSColorWell. NSColor supports a rich set of color formats and representations including custom ones. The other classes are mostly interface classes: They define and present panels and views that allow the user to select and apply colors. For example, the user can drag colors from the Color panel to any color well. The NSColorPicking protocol lets you extend the standard Color panel.

Dragging

With very little programming on your part, custom view objects can be dragged and dropped anywhere. Objects become part of this dragging mechanism by conforming to NSDragging... protocols: draggable objects conform to the NSDraggingSource protocol, and destination objects (receivers of a drop) conform to the NSDraggingDestination protocol. The Application Kit hides all the details of tracking the mouse and displaying the dragged image.

Printing and Faxing

The NSPrinter, NSPrintPanel, NSPageLayout, and NSPrintInfo classes work together to provide the means for printing and faxing the information that your application displays in its windows and views. You can also create an EPS representation of an NSView. This is easily done because the same representation, Postscript, is used for printing, faxing, and displaying.

Accessing the File System

Use the NSFileWrapper class to create objects that correspond to files or directories on disk. NSFileWrapper will hold the contents of the file in memory so that it can be displayed, changed, or transmitted to another application. It also provides an icon for dragging the file or representing it as an attachment (see "Text and Fonts"). Or use the NSFileManager class in the Foundation Framework to access and enumerate file and directory contents. The NSOpenPanel and NSSavePanel classes also provide a convenient and familiar user interface to the file system.

Sharing Data with Other Applications

The NSPasteboard class defines the pasteboard, a repository for data that's copied from your application, making this data available to any application that cares to use it. NSPasteboard implements the familiar cut-copy-paste operation. The NSServicesRequest protocol uses the pasteboard to communicate data that's passed between applications by a registered service.

Spell-Checking

The NSSpellServer class lets you define a spell-checking service and provide it as a service to other applications. To connect your application to a spell-checking service, you use the NSSpellChecker class. The NSIgnoreMisspelledWords and NSChangeSpelling protocols support the spell-checking mechanism.

Localization

If an application is to be used in more than one part of the world, its resources may need to be customized, or "localized," for language, country, or cultural region. For example, an application may need to have separate Japanese, English, French, and German versions of character strings, icons, nib files, or context help. Resource files specific to a particular language are grouped together in a subdirectory of the bundle directory (the directories with the ".lpro**j**" extension). Usually you setup localization resource files using Interface Builder. See the specifications for NSBundleAdditions and NSBundle class for more information on localization (NSBundle is in the Foundation Framework).

NSActionCell

Inherits From: NSCell : NSObject

Conforms To: NSCoding, NSCopying (NSCell)
NSObject (NSObject)

Declared In: AppKit/NSActionCell.h

Class Description

An NSActionCell defines an active area inside a control (an instance of NSControl or one of its subclasses). As an NSControl's active area, an NSActionCell does three things: it usually performs display of text or an icon; it provides the NSControl with a target and an action; and it handles mouse (cursor) tracking by properly highlighting its area and sending action messages to its target based on cursor movement. The only way to specify the NSControl for a particular NSActionCell is to send the NSActionCell a **drawWithFrame:inView:** message, passing the NSControl as the argument for the **inView:** keyword of the method.

NSActionCell implements the target object and action method as defined by its superclass, NSCell. As a user manipulates an NSControl, NSActionCell's **trackMouse:inRect:ofView:untilMouseUp:** method (inherited from NSCell) updates its appearance and sends the action message to the target object with the NSControl object as the only argument. See "Target and Action" below for more on this paradigm.

Usually, the responsibility for an NSControl's appearance and behavior is completely given over to a corresponding NSActionCell. (NSMatrix, and its subclass NSForm, are NSControls that don't follow this rule.)

A single NSControl may have more than one NSActionCell. To help identify it in this case, every NSActionCell has an integer tag. Note, however, that no checking is done by the NSActionCell object itself to ensure that the tag is unique. See the NSMatrix class for an example of a subclass of NSControl that contains multiple NSActionCells.

Many of the methods that define the contents and look of an NSActionCell, such as **setFont:** and **setBordered:**, are reimplementations of methods inherited from NSCell. They're overriden to ensure that the NSActionCell is redisplayed when "visual" attributes change.

Target and Action

Target objects and action methods (or messages) are part of the mechanism by which NSControls respond to user actions and enable users to communicate their intentions to an application. A target is an object that an NSControl uses as the receiver of action messages. The target's class defines an action method to enable its instances to respond to these messages, which are sent as users click or otherwise manipulate the

NSControl. NSControl's **sendAction:to:** asks the NSApplication object, NSApp, to send an action message to the NSControl's target object.

An action method takes only one argument: the **id** of the sender. The sender may be either the NSControl that sends the action message or, on occassion, another object that the target should treat as the sender. When it receives an action message, a target can return messages to the sender requesting additional information about its status.

You can also set the target to **nil** and allow it to be determined at run time. When the target is **nil**, the NSApplication object must look for an appropriate receiver. It conducts its search in a prescribed order, by following the responder chain until it finds an object that can respond to the message:

- It begins with the first responder in the key window and follows **nextResponder** links up the responder chain to the NSWindow's content view.
- It tries the NSWindow object and then the NSWindow's delegate.
- If the main window is different from the key window, it then starts over with the first responder in the main window and works its way up the main window's responder chain to the NSWindow object and its delegate.
- Next, the NSApplication object tries to respond itself. If it can't respond, it tries its own delegate. NSApp and its delegate are the receivers of last resort.

NSControl provides methods for setting and using the target object and the action method. However, these methods require that an NSControl's cell (or cells) be NSActionCells or custom cells that hold action and target as instance variables and can respond to the NSControl methods.

Method Types

Configuring an NSActionCell

– setAlignment:
– setBezeled:
– setBordered:
– setEnabled:
– setFloatingPointFormat:left:right:
– setFont:
– setImage:

Obtaining and setting cell values

– doubleValue
– floatValue
– intValue
– stringValue
– setObjectValue:

Displaying the NSActionCell

– drawWithFrame:inView:
– controlView

Assigning target and action

– setAction:
– action
– setTarget:
– target

Assigning a tag

– setTag:
– tag

Instance Methods

action

– (SEL)**action**

Returns the NSActionCell's action-message selector.

See also: – **setAction:**, – **setTarget:**, – **target**

controlView

– (NSView *)**controlView**

Returns the view (normally an NSControl) in which the NSActionCell was last drawn or **nil** if the NSActionCell has no control view (usually because it hasn't yet been placed in the view hierarchy).

See also: – **drawWithFrame:inView:**

doubleValue

– (double)**doubleValue**

Returns the NSActionCell's value as a **double** after validating any editing of cell content. If the receiver is not a text-type cell or the cell value is not scannable, the method returns zero.

See also: – **validateEditing** (NSControl)

drawWithFrame:inView:

– (void)**drawWithFrame:**(NSRect)*cellFrame* **inView:**(NSView *)*controlView*

Draws the NSActionCell's regular or bezeled border (if those attributes are set) and then draws the interior of the cell. NSActionCell's method overrides this method to replace its controlling control with *controlView* (if they're different) before invoking NSCell's **drawWithFrame:inView:**.

See also: – **controlView**

floatValue

– (float)**floatValue**

Returns the NSActionCell's value as a **float** after validating any editing of cell content. If the receiver is not a text-type cell or the cell value is not scannable, the method returns zero.

See also: – **validateEditing** (NSControl)

intValue

– (int)**intValue**

Returns the NSActionCell's value as a **int** after validating any editing of cell content. If the receiver is not a text-type cell or the cell value is not scannable, the method returns zero.

See also: – **validateEditing** (NSControl)

setAction:

– (void)**setAction:**(SEL)*aSelector*

Sets the selector used for the action message to *aSelector*.

See also: – **action**, – **setTarget:**, – **target**

setAlignment:

– (void)**setAlignment:**(NSTextAlignment)*mode*

Sets the alignment of text in the receiving NSActionCell; *mode* is one of five constants: NSLeftTextAlignment, NSRightTextAlignment, NSCenterTextAlignment, NSJustifiedTextAlignment, NSNaturalTextAlignment (the default alignment for the text). The method marks the receiving NSActionCell as needing redisplay after discarding any editing changes that were being made to cell text.

setBezeled:

– (void)**setBezeled:**(BOOL)*flag*

Sets whether the NSActionCell draws itself with a bezeled border and marks it as needing redisplay. The **setBezeled:** and **setBordered:** methods are mutually exclusive (that is, a border can be only plain or bezeled).

setBordered:

– (void)**setBordered:**(BOOL)*flag*

Sets whether the receiver draws itself outlined with a plain border and marks it as needing redisplay. The **setBezeled:** and **setBordered:** methods are mutually exclusive (that is, a border can be only plain or bezeled).

setEnabled:

– (void)**setEnabled:**(BOOL)*flag*

Sets whether the receiver is enabled or disabled. The text of disabled cells is changed to gray. If a cell is disabled, it cannot be highlighted, does not support mouse tracking (and thus cannot participate in target/action functionality), and cannot be edited. The method marks the receiving NSActionCell as needing redisplay after discarding any editing changes that were being made to cell text.

setFloatingPointFormat:left:right:

– (void)**setFloatingPointFormat:**(BOOL)*autoRange*
left:(unsigned int)*leftDigits*
right:(unsigned int)*rightDigits*

Sets the NSActionCell's floating point format as described in the NSCell class specification for the **setFloatingPointFormat:left:right:** method. NSActionCell's implementation of the method supplements NSCell's by marking the receiving NSActionCell as needing redisplay after discarding any editing changes that were being made to cell text.

setFont:

– (void)**setFont:**(NSFont *)*fontObj*

Sets the font to be used when the NSActionCell displays text. If the receiver is not a text-type cell, the method converts it to that type. If *fontObj* is **nil** and the receiver is a text-type cell, the font currently held by the receiver is autoreleased. NSActionCell supplements NSCell's implementation of this method by

marking the updated cell as needing redisplay; if the receiving NSActionCell was converted to a text-type cell and is selected, it also updates the field editor with *fontObj*.

setImage:

– (void)**setImage:**(NSImage *)*image*

Sets the image to be displayed in the receiver. If *image* is **nil**, the image currently displayed by the receiver is removed.

setObjectValue:

– (void)**setObjectValue:**(id)*object*

Discards any editing of the receiving NSActionCell's text and sets its object value to *object*. If the object value is afterwards different from what it was before the method was invoked, the method marks the NSActionCell as needing redisplay.

setTag:

– (void)**setTag:**(int)*anInt*

Sets the receiving NSActionCell's tag to *anInt*.

See also: – **tag**

setTarget:

– (void)**setTarget:**(id)*anObject*

Sets the receiving NSActionCell's target object to *anObject*.

See also: – **action**, – **setAction:**, – **target**

stringValue

– (NSString *)**stringValue**

Returns the receiving NSActionCell's value as a string object as converted by the cell's formatter, if one exists. If no formatter exists and the value is an NSString, returns the value as an plain, attributed or localized formatted string. If the value is not an NSString or can't be converted to one, returns an empty

string. The method supplements NSCell's implementation by validating and retaining any editing changes being made to cell text.

See also: – **validateEditing** (NSControl)

tag

– (int)**tag**

Returns the receiving NSActionCell's tag.

See also: – **setTag:**

target

– (id)**target**

Returns the receiving NSActionCell's target object.

See also: – **action**, – **setAction:**, – **setTarget:**

NSAffineTransform

Inherits From:	NSObject
Conforms To:	NSCoding NSCopying NSObject (NSObject)
Declared In:	AppKit/NSAffineTransform.h

Class Description

The NSAffineTransform class provides methods for creating, concatenating, and applying affine transformations. A *transformation* specifies how points in one coordinate system are transformed to points in another coordinate system. An *affine transformation* is a special type of transformation that preserves parallel lines in a path but does not necessarily preserve lengths or angles. Scaling, rotation, and translation are the most commonly used manipulations supported by affine trasforms, but shearing is also possible.

Graphic transformations rely on a matrix of values to specify how a path will be changed. Points being transformed are multiplied through the matrix to obtain the resulting (transformed) point. You do not need to know anything about matrices and matrix multiplication to use the NSAffineTransform class. NSAffineTransform provides methods that allow you to set scaling, rotation, and translation factors while leaving the matrix mathematics to the transform object.

Creating Transforms

To create a new instance of NSAffineTransform, you can use either the **transform** class method or you can allocate a new transform object and initialize it using the **initWithTransform:** method. When you use the **transform** class method to create a new instance of NSAffineTransform, the new object's matrix is initialized to the identity matrix (which transforms any point to itself). When you use the **initWithTransform:** method, the new object's matrix is set to the matrix of the specified affine transform.

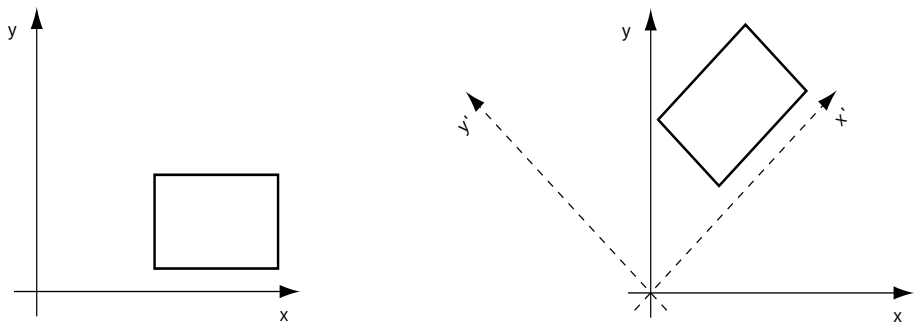
NSAffineTransform provides direct support for setting scaling, rotation, and translation factors. Use the **translateXBy:yBy:** method to set translation factors independently for each axis. Use the **rotateByDegrees:** and **rotateByRadians:** methods to set rotation factors either in degrees or in radians. Use the **scaleBy:** method to set the same scaling factor for both x and y axes, or use the **scaleXBy:yBy:** method to set independent scaling factors for each axis. If you want to set a combination of factors all at once (or if you want to set shearing factors), you can use the **setTransformStruct:** method to specify the matrix values directly.

For more information on the mathematics involved in transform manipulations, see "Manipulating Transform Values".

Using Transforms

Transforms are used most commonly to adjust a view's drawing area by translating the view's origin to some prescribed location on the screen. However, you can also use transforms to arbitrarily scale or rotate paths within their view. Because transformations occur relative to the origin of the local coordinate system, several transforms may need to be concatenated together to generate the proper effect. For example, Figure 1 shows a rectangular path and the result of rotating it 45 degrees. In order to rotate the rectangle around its origin, you would need to translate that origin to the center of its coordinate system, apply the rotation, and then translate the rectangle back to its original location. You may need to apply similar techniques when scaling a path.

Figure 1 Rotating a rectangular path 45 degrees



a) Unmodified rectangle

b) Rectangle rotated 45°

Rather than applying separate transforms to translate, scale, and rotate a single path you can combine a group of transforms into a single transform and apply only that transform. To do this, you must combine the transforms by appending them together with the **appendTransform:** method. The effects of the appended transform occur after the effects of the original transform. Similarly, you can prepend one transform to another using the **prependTransform:** method.

Once you have constructed your transform object and set the appropriate scaling, translation, and rotation factors, you can apply the transformation to one or more points. To apply the transform to a single point, use the **transformPoint:** method. To transform an entire path, use the **transformBezierPath:** method, which returns a copy of the transformed path leaving the original untouched. You can also transform NSSize values using the **transformSize:** method, although NSSize values are not affected by translations.

You can also apply a transform to all of the current drawing operations using the **concat** method. This method appends the transform's matrix to the transformation matrix stored in the current graphic context.

Note: By the time the **drawRect:** method is invoked, the current transformation is already a concatenation of the screen's, window's and any superview's transformations. Concatenating your transformation to the current transformation modifies subsequent drawing operations within the bounds of your NSView object.

To cancel the effects of a matrix, simply invoke its **invert** method and apply it again.

Manipulating Transform Values

An NSAffineTransform object uses a 3x3 transformation matrix of the form shown in Figure

Figure 2 A 3x3 matrix

$$\begin{bmatrix} m_{11} & m_{12} & 0 \\ m_{21} & m_{22} & 0 \\ t_x & t_y & 1 \end{bmatrix}$$

where a point (x,y) is transformed into another point (x',y') using these linear equations:

```
x' = (m11)x + (m21)y + tx
y' = (m12)x + (m22)y + ty
```

Concatenation, translation, rotation, and scaling are performed by matrix multiplication. The order in which transformations are multiplied is important because matrix operations are associative, but not commutative (matrix1 $\times$ matrix2 $\neq$ matrix2 $\times$ matrix1).

You can specify the matrix values directly using the NSAffineTransformStruct, which lets you specify the six values for *m11*, *m12*, *m21*, *m22*, *tx*, and *ty*. You can then use the **setTransformStruct:** method to associate your matrix values with the transform object. FigureXX shows the standard set of matrices used to implement different effects. You can multiply these matrices together to obtain the formulae for a

complex set of translations, scales, rotations, and shears. For translations, scales, and shears, the subscripts of matrix elements indicate the axis affected by values at that location.

Figure 3 Matrices for translation, rotation, scaling, and shearing

$$T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix}$$

a) Translation matrix

$$R(\Theta) = \begin{bmatrix} \cos\Theta & \sin\Theta & 0 \\ -\sin\Theta & \cos\Theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

b) Rotation matrix (with Θ measured in radians)

$$S = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

c) Scaling matrix

$$SH = \begin{bmatrix} 1 & sh_y & 0 \\ sh_x & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

d) Shearing matrix

Adopted Protocols

NSCoding

– encodeWithCoder:
– initWithCoder:

NSCopying

– copyWithZone:

Method Types

Creating an NSAffineTransform object

+ transform
– initWithTransform:

Accumulating Transformations

– rotateByDegrees:
– rotateByRadians:
– scaleBy:
– scaleXBy:yBy:
– translateXBy:yBy:
– appendTransform:
– prependTransform:
– invert

Setting the Current Transform in the Current Graphics State

– set
– concat

Transforming Data and Objects

– transformBezierPath:
– transformPoint:
– transformSize:

Transformation Struct

– transformStruct
– setTransformStruct:

Class Methods

transform

+ (NSAffineTransform *)**transform**

Creates and returns a new instance of NSAffineTransform initialized to the identity matrix. The *identity matrix* transforms any point to itself.

See also: – **initWithTransform:**

Instance Methods

appendTransform:

– (void)**appendTransform:**(NSAffineTransform *)*aTransform*

Performs a multiplication of the receiver's matrix and *aTransform*'s matrix, and replaces the receiver's matrix with the result. This has the effect of applying *aTransform*'s matrix after the receiver's matrix.

See also: – **prependTransform:**

concat

– (void)**concat**

Appends the receiver's matrix to the current transformation matrix stored in the current graphics context replacing the current transformation matrix with the result (concatenation is performed by matrix multiplication, see Class Description above).

If this method is invoked from within an NSView's **drawRect:** method, then the current transformation matrix is an accumulation of the screen, window and any superview's transformation matrices. Invoking this method defines a new user coordinate system whose coordinates are mapped into the former coordinate system according to the receiver's transformation matrix. To undo the concatenation, you must invert the receiver's matrix and invoke this method again.

See also: **– set**, **– invert**

initWithTransform:

– (id)**initWithTransform:**(NSAffineTransform *)*aTransform*

Initializes the receiver's matrix to the matrix in *aTransform* and returns the receiver.

See also: + **transform**

invert

– (void)**invert**

Replaces the receiver's matrix with its inverse matrix. Inverse matrices are useful for undoing the effects of a matrix. If a previous point (x,y) was transformed to (x',y'), inverting the matrix and applying it to point (x',y') will yield the point (x,y).

You can also use them in conjunction with the **concat** method to remove the effects of concatenating the matrix to the current transformation matrix of the current graphic context.

See also: **– concat**

prependTransform:

– (void)**prependTransform:**(NSAffineTransform *)*aTransform*

Prepends the matrix of *aTransform* to the receiver's matrix. Multiplies *aTransform* and the receiver's matrix, and replaces the receiver's matrix with the result. This has the effect of applying *aTransform* before the receiver's transform.

See also: – **appendTransform:**

rotateByDegrees:

– (void)**rotateByDegrees:**(float)*angle*

Replaces the receiver's transformation matrix with a matrix thar applies a rotation of *angle* degrees. Any previous transformations associated with the matrix are lost and replaced by the rotation.

After invoking this method, applying the receiver's matrix will turn the axes counterclockwise about the current origin by *angle* degrees.

See also: – **rotateByRadians:**, – **scaleBy:**, **– scaleXBy:yBy:**, – **translateXBy:yBy:**

rotateByRadians:

– (void)**rotateByRadians:**(float)*angle*

Replaces the receiver's transformation matrix with a matrix thar applies a rotation of *angle* radians. Any previous transformations associated with the matrix are lost and replaced by the rotation.

After invoking this method, applying the receiver's matrix will turn the axes counterclockwise about the current origin by *angle* radians.

See also: – **rotateByDegrees:**, – **scaleBy:**, **– scaleXBy:yBy:**, – **translateXBy:yBy:**

scaleBy:

– (void)**scaleBy:**(float)*scale*

Replaces the receiver's transformation matrix with a matrix that applies the specified scaling factor along both x and y axes. Any previous transformations associated with the matrix are lost and replaced by the new scaling factor.

Hereafter, applying the receiver's matrix will modify the unit lengths along the current *x* and *y* axes by a factor of *scale*.

See also: – **rotateByDegrees:**, – **rotateByRadians:**, – **scaleXBy:yBy:, – translateXBy:yBy:**

scaleXBy:yBy:

– (void)**scaleXBy:**(float)*scaleX* **yBy:**(float)*scaleY*

Replaces the receiver's transformation matrix with a matrix that applies the *scaleX* factor along the x axis and the *scaleY* factor along the y axis. Any previous transformations associated with the matrix are lost and replaced by the new scaling factors.

After invoking this method, applying the receiver's matrix will modify the unit length on the *x* axis by a factor of *scaleX* and the *y* axes by a factor of *scaleY.*

See also: – **rotateByDegrees:**, – **rotateByRadians:**, – **scaleBy:**, **– translateXBy:yBy:**

set

– (void)**set**

Sets the current transformation matrix to the receiver's transformation matrix. The *current transformation* is stored in the current graphics context and will be applied to subsequent drawing operations. You should use this method sparingly because it removes the existing transformation matrix, which is an accumulation of transformation matrices for the screen, window and any superviews. Instead use the **concat** method to add this transformation matrix to the current transformation matrix.

setTransformStruct:

– (void)**setTransformStruct:**(NSAffineTransformStruct)*aTransformStruct*

Sets the receiver's transformation matrix using the values in *aTransformStruct* where the matrix is of the form shown in Figure 2 and the six-element structure defined by an NSAffineTransformStruct is of the form:

```
{m11, m12, m21, m22, tx, ty}
```

NSAffineTransformStruct is an alternate representation of a transformation matrix that can be used to specify matrix values directly.

See also: – **initWithTransform:**, – **transformStruct**

transformBezierPath:

– (NSBezierPath *)**transformBezierPath:**(NSBezierPath *)*aPath*

Creates and returns a new NSBezierPath object with each point in *aPath* transformed by the receiver. The original *aPath* is not modified.

See also: – **transformPoint:**, – **transformSize:**

transformPoint:

– (NSPoint)**transformPoint:**(NSPoint)*aPoint*

Returns the result of applying the receiver's transform to *aPoint*.

See also: – **transformBezierPath:**, – **transformSize:**

transformSize:

– (NSSize)**transformSize:**(NSSize)*aSize*

Returns the result of applying the receiver's transform to *aSize*. Since *aSize* specifies a width and height, not an *x* and *y* coordinate, translation factors are not applied. This method is useful for transforming delta or distance values when you need to take scaling factors into account

See also: – **transformBezierPath:**, – **transformPoint:**

transformStruct

– (NSAffineTransformStruct)**transformStruct**

Returns the NSAffineTransformationStruct equivalent to the receiver's matrix where the matrix is of the form shown in Figure 2 and the six-element structure defined by an NSAffineTransformStruct is of the form:

```
{m11, m12, m21, m22, tx, ty}
```

NSAffineTransformStruct is an alternate representation of a transformation matrix that can be used to specify matrix values directly.

See also: – **initWithTransform:**, – **setTransformStruct:**

translateXBy:yBy:

– (void)**translateXBy:**(float)*deltaX* **yBy:**(float)*deltaY*

Replaces the receiver's transformation matrix with a matrix that applies the specified translation factors. Subsequent transformations will cause coordinates to be shifted by *deltaX* units along the x-axis and by *deltaY* units along the y-axis. Any previous transformations associated with the matrix are lost and replaced by the new scaling factors.

Translation factors do not affect NSSize values, which specify a differential between points.

See also: – **rotateByDegrees:**, – **rotateByRadians:**, – **scaleBy:**, **– scaleXBy:yBy:**

NSApplication

Inherits From:	NSResponder : NSObject
Conforms To:	NSCoding (NSResponder) NSObject (NSObject)
Declared In:	AppKit/NSApplication.h AppKit/NSColorPanel.h AppKit/NSDataLinkPanel.h AppKit/NSHelpManager.h AppKit/NSPageLayout.h

Class at a Glance

Purpose

An NSApplication object manages an application's main event loop in addition to resources used by all of that application's objects.

Principal Attributes

- Delegate
- Key window
- DPS context
- List of windows
- Main window

Creation

Project Builder

+ sharedApplication	Creates the shared application instance (global variable NSApp).

Commonly Used Methods

– keyWindow	Returns an NSWindow representing the key window.
– mainWindow	Returns an NSWindow representing the main window.
– registerServicesMenuSendTypes:returnTypes:	Specifies which services are valid for this application.
– runModalForWindow:	Runs a modal event loop for the specified NSWindow.

Class Description

The NSApplication class provides the central framework for your application's execution. Every application must have exactly one instance of NSApplication (or a subclass of NSApplication). Your program's **main()** function should create this instance by invoking the **sharedApplication** class method. After creating the NSApplication object, the **main()** function should load your application's main nib file and then start the event loop by sending the NSApplication object a **run** message. If you create an Application project in Project Builder, this **main()** function is created for you. The **main()** function that

Project Builder creates begins by calling a function named **NSApplicationMain()**, which is functionally similar to the following:

```
void NSApplicationMain(int argc, char *argv[]) {
    [NSApplication sharedApplication];
    [NSBundle loadNibNamed:@"myMain" owner:app];
    [NSApp run];
}
```

The **sharedApplication** class method initializes the PostScript environment and connects your program to the Window Server and the Display PostScript server. The NSApplication object maintains a list of all the NSWindows that the application uses, so it can retrieve any of the application's NSViews. **sharedApplication** also initializes the global variable NSApp, which you use to retrieve the NSApplication instance. **sharedApplication** only performs the initialization once; if you invoke it more than once, it simply returns the NSApplication object that it created previously.

NSApplication's main purpose is to receive events from the Window Server and distribute them to the proper NSResponders. NSApp translates an event into an NSEvent object, then forwards the NSEvent to the affected NSWindow object. All keyboard and mouse events go directly to the NSWindow associated with the event. The only exception to this rule is if the Command key is pressed when a key-down event occurs; in this case, every NSWindow has an opportunity to respond to the event. When an NSWindow receives an NSEvent from NSApp, it distributes it to the objects in its view hierarchy.

The NSApplication class sets up autorelease pools (instances of the NSAutoreleasePool class) during initialization and inside the event loop—specifically, within its **init** (or **sharedApplication**) and **run** methods. Similarly, the methods that the Application Kit adds to NSBundle employ autorelease pools during the loading of nib files. These autorelease pools aren't accessible outside the scope of the respective NSApplication and NSBundle methods. Typically, an application creates objects either while the event loop is running or by loading objects from nib files, so this usually isn't a problem. However, if you do need to use OpenStep classes within the **main()** function itself (other than to load nib files or to instantiate NSApplication), you should create an autorelease pool before using the classes and then release the pool when you're done. For more information, see the NSAutoreleasePool class specification in the *Foundation Framework Reference*.

Subclassing NSApplication

Rarely do you need to create a custom NSApplication subclass. In general, a better design is to separate the code that embodies your program's functionality into a number of custom objects. Usually, those custom objects are subclasses of NSObject. Methods defined in your custom objects can be invoked from a small dispatcher object without being closely tied to NSApp. The only reason to subclass NSApplication is if you need to provide your own special response to messages that are routinely sent to NSApp. (Even then, NSApp's delegate is often given a chance to respond to such messages, so it's more appropriate to implement the delegate methods.) To use a custom subclass of NSApplication, simply send **sharedApplication** to your custom class rather than directly to NSApplication. If you create your application in Project Builder, set the application class on the Project Attributes inspector, and Project

Builder will update the **main()** function accordingly. As mentioned previously, NSApp uses autorelease pools in its **init** and **run** methods; if you override these methods, you'll need to create your own autorelease pools.

The Delegate and Notifications

You can assign a delegate to NSApp. The delegate responds to certain messages on behalf of NSApp. Some of these messages, such as **application:openFile:**, ask the delegate to open a file. Another message, **applicationShouldTerminate:**, lets the delegate determine whether the application should be allowed to quit. The NSApplication class sends these messages directly to its delegate.

NSApp also posts notifications to the application's default notification center. Any object may register to receive one or more of the notifications posted by NSApp by sending the message **addObserver:selector: name:object:** to the default notification center (an instance of the NSNotificationCenter class). NSApp's delegate is automatically registered to receive these notifications if it implements certain delegate methods. For example, NSApp posts notifications when it is about to be done launching the application and when it is done launching the application (NSApplicationWillFinishLaunchingNotification and NSApplicationDidFinishLaunchingNotification). The delegate has an opportunity to respond to these notifications by implementing the methods **applicationWillFinishLaunching:** and **applicationDidFinishLaunching:**. If the delegate wants to be informed of both events, it implements both methods. If it only needs to know when the application is finished launching, it implements only **applicationDidFinishLaunching:**. For more information on notifications, see the NSNotificationCenter class specification in the *Foundation Framework Reference*.

Method Types

Creating and initializing an NSApplication

+ sharedApplication
– finishLaunching

Changing the active application

– activateIgnoringOtherApps:
– isActive
– deactivate

Running the event loop

– run
– isRunning
– stop:
– runModalForWindow:
– stopModal
– stopModalWithCode:
– abortModal
– beginModalSessionForWindow:
– runModalSession:
– endModalSession:
– sendEvent:

Getting, removing, and posting events

– currentEvent
– nextEventMatchingMask:untilDate:inMode:dequeue:
– discardEventsMatchingMask:beforeEvent:
– postEvent:atStart:

Managing windows

– keyWindow
– mainWindow
– windowWithWindowNumber:
– windows
– makeWindowsPerform:inOrder:
– setWindowsNeedUpdate:
– updateWindows
– miniaturizeAll:
– preventWindowOrdering

Hiding all windows

– hide:
– isHidden
– unhide:
– unhideWithoutActivation

Setting the application’s icon

– setApplicationIconImage:
– applicationIconImage

Getting the main menu

– setMainMenu:
– mainMenu

Managing the Window menu

– setWindowsMenu:
– windowsMenu
– arrangeInFront:
– addWindowsItem:title:filename:
– changeWindowsItem:title:filename:
– removeWindowsItem:
– updateWindowsItem:

Managing the Services menu

– setServicesMenu:
– servicesMenu
– registerServicesMenuSendTypes:returnTypes:
– validRequestorForSendType:returnType:
– setServicesProvider:
– servicesProvider

Showing standard panels

– orderFrontColorPanel:
– orderFrontDataLinkPanel:
– runPageLayout:

Displaying help

– showHelp:
– activateContextHelpMode:

Sending action messages

– sendAction:to:from:
– tryToPerform:with:
– targetForAction:

Getting the Display PostScript context

– context

Reporting an exception

– reportException:

Terminating the application

– terminate:

Assigning a delegate

– setDelegate:
– delegate

Microsoft Windows® specific methods

– applicationHandle
– windowWithWindowHandle:
+ setApplicationHandle:previousHandle:commandLine:show:
+ useRunningCopyOfApplication

Class Methods

setApplicationHandle:previousHandle:commandLine:show:

+ (void)**setApplicationHandle:**(void *)*hInstance*
previousHandle:(void *)*prevInstance*
commandLine:(NSString *)*cmdLine*
show:(int)*cmdShow*

On Microsoft Windows platforms, informs the NSApplication class of the values for the arguments passed to the **WinMain()** function. This message should be sent once, as the first line of the **WinMain()** function. If you create your application using Project Builder, this is done for you. You only need to invoke this method if you implement your own **WinMain()** function. Don't override this method in NSApplication subclasses.

This method is not implemented on the Mach platform.

See also: – **applicationHandle**

sharedApplication

+ (NSApplication *)**sharedApplication**

Returns the NSApplication instance (the global NSApp), creating it if it doesn't exist yet. This method also makes a connection to the Window Server and completes other initialization. Your program should invoke this method as one of the first statements in **main()**; this is done for you if you create your application with Project Builder. To retrieve the NSApplication instance after it has been created, you use the global variable NSApp or invoke this method.

See also: – **run**, – **terminate:**

useRunningCopyOfApplication

+ (void)**useRunningCopyOfApplication**

On Microsoft Windows platforms, attempts to find an already running copy of the application at startup. This method is invoked in the **WinMain()** function. If the command used to start the application contains the option **-NSUseRunningCopy YES** and the application is already running, this method causes that version of the application to be activated rather than start up a new copy.

The method returns if the **-NSUseRunningCopy YES** option was not specified, if there was no previously running copy, or if the running copy was unable to be used (for any reason). If a running copy is successfully found and used, this method exits with a code of 0.

You never need to invoke this method directly. If you need to prevent the system from using an already running copy of the application, write your own **WinMain()** function, removing this method invocation. NSApplication subclasses should not override this method.

This method is not defined for the Mach platform.

Instance Methods

abortModal

– (void)**abortModal**

Aborts the event loop started by **runModalForWindow:** by raising an NSAbortModalException, which is caught by **runModalForWindow:**. Because this method raises an exception, it never returns; **runModalForWindow:**, when stopped with this method, returns NSRunAbortedResponse. **abortModal** is typically sent by objects registered with the default NSRunLoop; for example, by objects that have registered a method to be repeatedly invoked by the NSRunLoop through the use of an NSTimer object.

This method can also abort a modal session created by **beginModalSessionForWindow:**, provided the loop that runs the modal session (by invoking **runModalSession:**) catches NSAbortModalException.

See also: – **endModalSession:**, – **stopModal**, – **stopModalWithCode:**

activateContextHelpMode:

– (void)**activateContextHelpMode:**(id)*sender*

Places the application in context-sensitive help mode. In this mode, the cursor becomes a question mark, and help appears for any user interface item that the user clicks. This method is typically invoked on Microsoft Windows platforms when the user selects the What's This menu item. (An application also enters context-sensitive help mode on Microsoft Windows platforms when the user presses Shift-F1.)

On Mach platforms, most applications don't use this method. Instead, applications enter context-sensitive mode when the user presses the Help key. On either platform, applications exit context-sensitive help mode upon the first event after a help window is displayed.

See also: – **showHelp:**

activateIgnoringOtherApps:

– (void)**activateIgnoringOtherApps:**(BOOL)*flag*

Makes the receiver the active application. If *flag* is NO, the application is activated only if no other application is currently active. If *flag* is YES, the application activates regardless.

On Mach platforms, *flag* is normally set to NO. When the Workspace Manager launches an application, using a value of NO for *flag* allows the application to become active if the user waits for it to launch, but the application remains unobtrusive if the user activates another application. Regardless of the setting of

flag, there may be a time lag before the application activates; you should not assume that the application will be active immediately after sending this message.

On Microsoft Windows platforms, *flag* is normally set to YES. Setting *flag* to NO has no effect.

You rarely need to invoke this method. Under most circumstances, the Application Kit takes care of proper activation. However, you might find this method useful if you implement your own methods for interapplication communication.

You don't need to send this message to make one of the application's NSWindows key. When you send a **makeKeyWindow** message to an NSWindow, you simply ensure that the NSWindow will be the key window when the application is active.

See also: – **deactivate**, – **isActive**

addWindowsItem:title:filename:

– (void)**addWindowsItem:**(NSWindow *)*aWindow*
title:(NSString *)*aString*
filename:(BOOL)*isFilename*

Adds an item to the Window menu for *aWindow*. If *isFilename* is NO, *aString* appears literally in the menu. If *isFilename* is YES, *aString* is assumed to be a converted path name with the name of the file preceding the path (the way NSWindow's **setTitleWithRepresentedFilename:** method shows a title). If an item for *aWindow* already exists in the Window menu, this method has no effect. You rarely invoke this method because an item is placed in the Window menu for you whenever an NSWindow's title is set.

See also: – **changeWindowsItem:title:filename:**, – **setTitle:** (NSWindow)

applicationHandle

- (void *)**applicationHandle**

On Microsoft Windows platforms, returns the application's Win32 instance handle, which is a required parameter for some Win32 function calls. This method is not defined for the Mach platform.

See also: + **setApplicationHandle:previousHandle:commandLine:show:**

applicationIconImage

– (NSImage *)**applicationIconImage**

Returns the NSImage used for the application's icon, which represents the application in the Workspace Manager on Mach platforms or in the Program Manager on Microsoft Windows platforms.

See also: – **setApplicationIconImage:**

arrangeInFront:

– (void)**arrangeInFront:**(id)*sender*

Arranges all of the windows listed in the Window menu in front of all other windows. Windows associated with the application but not listed in the Window menu are not ordered to the front.

See also: – **addWindowsItem:title:filename:**, – **removeWindowsItem:**, – **makeKeyAndOrderFront:** (NSWindow)

beginModalSessionForWindow:

– (NSModalSession)**beginModalSessionForWindow:**(NSWindow *)*aWindow*

Sets up a modal session with the NSWindow *aWindow* and returns an NSModalSession structure representing the session. In a modal session, the application receives mouse events only if they occur in *aWindow*. The NSWindow is made key and ordered to the front.

beginModalSessionForWindow: only sets up the modal session. To actually run the session, use **runModalSession:**. **beginModalSessionForWindow:** should be balanced by **endModalSession:**. Make sure that these two messages are sent within the same exception handling scope. That is, if you send **beginModalSessionForWindow:** inside of an **NS_DURING** construct, you must send **endModalSession:** before **NS_ENDHANDLER**.

If an exception is raised, **beginModalSessionForWindow:** arranges for proper cleanup. Do *not* use NS_DURING constructs to send an **endModalSession:** message in the event of an exception.

A loop using these methods is similar to a modal event loop run with **runModalForWindow:**, except that the application can continue processing between method invocations.

changeWindowsItem:title:filename:

– (void)**changeWindowsItem:**(NSWindow *)*aWindow* **title:**(NSString *)*aString* **filename:**(BOOL)*isFilename*

Changes the item for *aWindow* in the Window menu to *aString*. If *aWindow* doesn't have an item in the Window menu, this method adds the item. If *isFilename* is NO, *aString* appears literally in the menu. If *isFilename* is YES, *aString* is assumed to be a converted path name with the file's name preceding the path (the way NSWindow's **setTitleWithRepresentedFilename:** places a title).

See also: – **addWindowsItem:title:filename:**, – **removeWindowsItem:**, – **setTitle:** (NSWindow), – **setTitleWithRepresentedFilename:** (NSWindow)

context

– (NSDPSContext *)**context**

Returns the receiver's Display PostScript context.

currentEvent

– (NSEvent *)**currentEvent**

Returns the current event, the last event the receiver retrieved from the event queue. NSApp receives events and forwards the current event to the affected NSWindow object, which then distributes it to the objects in its view hierarchy.

See also: – **discardEventsMatchingMask:beforeEvent:**, – **postEvent:atStart:**, – **sendEvent:**

deactivate

– (void)**deactivate**

Deactivates the application. Normally, you shouldn't invoke this method; the Application Kit is responsible for proper deactivation.

See also: – **activateIgnoringOtherApps:**

delegate

– (id)**delegate**

Returns the receiver's delegate.

See also: – **setDelegate:**

discardEventsMatchingMask:beforeEvent:

– (void)**discardEventsMatchingMask:**(unsigned int)*mask* **beforeEvent:**(NSEvent *)*lastEvent*

Removes from the event queue all events matching those specified in *mask* that were generated before *lastEvent*. Typically, you send this message to an NSWindow rather than to NSApp.

mask can contain these constants:

Constant	Description
NSLeftMouseDownMask	The left mouse button was pressed.

Constant	Description
NSLeftMouseUpMask	The left mouse button was released.
NSRightMouseDownMask	The right mouse button was pressed.
NSRightMouseUpMask	The right mouse button was released.
NSMouseMovedMask	The user moved the mouse.
NSLeftMouseDraggedMask	The user moved the mouse while the left button was pressed.
NSRightMouseDraggedMask	The user moved the mouse while the right button was pressed.
NSMouseEnteredMask	The mouse entered a tracking rectangle.
NSMouseExitedMask	The mouse exited a tracking rectangle.
NSKeyDownMask	A key on the keyboard was pressed.
NSKeyUpMask	A key on the keyboard was released.
NSFlagsChangedMask	A Shift, Command, Alternate, or Escape key was pressed or released.
NSPeriodicMask	A periodic event occurred.
NSCursorUpdateMask	Cursor update.
NSAnyEventMask	Any event.

Use this method to ignore certain events that occurred after a particular event. For example, suppose your application has a tracking loop that you exit when the user releases the mouse button, and you want to discard all of the events that occurred during that loop. You use NSAnyEvent as the *mask* argument and pass the mouse up event as the *lastEvent* argument. Passing the mouse-up event as *lastEvent* ensures that any events that might have occurred after the mouse-up event (that is, that appear in the queue after the mouse-up event) don't get discarded.

See also: – **nextEventMatchingMask:untilDate:inMode:dequeue:**

encodeWithCoder:

@protocol NSCoding
– (void)**encodeWithCoder:**(NSCoder *)*aCoder*

Raises an NSInvalidArgumentException. You cannot encode an NSApplication instance.

See also: – **initWithCoder:**

endModalSession:

– (void)**endModalSession:**(NSModalSession)*session*

Finishes a modal session. The argument *session* should be the return value from a previous invocation of **beginModalSessionForWindow:**.

See also: – **runModalSession:**

finishLaunching

– (void)**finishLaunching**

Activates the application, opens any files specified by the "NSOpen" user default, and unhighlights the application's icon. The **run** method invokes this method before it starts the event loop. When this method begins, it posts an NSApplicationWillFinishLaunchingNotification to the default notification center. When it successfully completes, it posts an NSApplicationDidFinishLaunchingNotification. If you override **finishLaunching**, the subclass method should invoke the superclass method.

See also: – **applicationWillFinishLaunching:** (delegate method), – **applicationDidFinishLaunching:** (delegate method)

hide:

– (void)**hide:**(id)*sender*

Hides all the application's windows. This method is usually invoked when the user chooses Hide in the application's main menu. When this method begins, it posts an NSApplicationWillHideNotification to the default notification center. When it completes successfully, it posts an NSApplicationDidHideNotification.

See also: – **applicationDidHide:** (delegate method), – **applicationWillHide:** (delegate method), – **miniaturizeAll:**, – **unhide:**, – **unhideWithoutActivation**

initWithCoder:

@protocol NSCoding
– (id)**initWithCoder:**(NSCoder *)*aDecoder*

Raises an NSInvalidArgumentException. You cannot encode an NSApplication instance.

See also: – **encodeWithCoder:**

isActive

– (BOOL)**isActive**

Returns YES if this is the active application, NO otherwise.

See also: – **activateIgnoringOtherApps:**, – **deactivate**

isHidden

– (BOOL)**isHidden**

Returns YES if the application is hidden, NO otherwise.

See also: – **hide:**, – **unhide:**, – **unhideWithoutActivation**

isRunning

– (BOOL)**isRunning**

Returns YES if the main event loop is running, NO otherwise. NO means the **stop:** method was invoked.

See also: – **run**, – **terminate:**

keyWindow

– (NSWindow *)**keyWindow**

Returns the key window, the NSWindow that receives keyboard events. This method returns **nil** if there is no key window, if the application's nib file hasn't finished loading yet, or if the key window belongs to another application.

See also: – **mainWindow**, – **isKeyWindow** (NSWindow)

mainMenu

– (NSMenu *)**mainMenu**

Returns the application's main menu.

See also: – **setMainMenu:**

mainWindow

– (NSWindow *)**mainWindow**

Returns the main window. This method returns **nil** if there is no main window, if the application's nib file hasn't finished loading, if the main window belongs to another application, or if the application is hidden.

See also: – **keyWindow**, – **isMainWindow** (NSWindow)

makeWindowsPerform:inOrder:

– (NSWindow *)**makeWindowsPerform:**(SEL)*aSelector* **inOrder:**(BOOL)*flag*

Sends the *aSelector* message to each NSWindow in the application in turn until one of them returns a value other than **nil**. Returns that NSWindow, or **nil** if all of the NSWindows returned **nil** for *aSelector*.

If *flag* is YES, the NSWindows receive the *aSelector* message in the front-to-back order in which they appear in the Window Server's window list. If *flag* is NO, NSWindows receive the message in the order they appear in NSApp's window list. This order is unspecified.

The method designated by *aSelector* can't take any arguments.

See also: – **sendAction:to:from:**, – **tryToPerform:with:**, – **windows**

miniaturizeAll:

– (void)**miniaturizeAll:**(id)*sender*

Miniaturizes all the receiver's windows.

See also: – **hide:**

nextEventMatchingMask:untilDate:inMode:dequeue:

– (NSEvent *)**nextEventMatchingMask:**(unsigned int)*mask*
untilDate:(NSDate *)*expiration*
inMode:(NSString *)*mode*
dequeue:(BOOL)*flag*

Returns the next event matching *mask*, or **nil** if no such event is found before the *expiration* date. If *flag* is YES, the event is removed from the queue. See the method description for **discardEventsMatchingMask: beforeEvent:** for a list of the possible values for *mask*.

The *mode* argument names an NSRunLoop mode that determines what other ports are listened to and what timers may fire while NSApp is waiting for the event. The possible modes available in the Application Kit are:

Mode	Description
NSDefaultRunLoopMode	Main event loop.
NSEventTrackingRunLoopMode	Modal event loops.
NSModalPanelRunLoopMode	Loops that operate while a modal panel is up.
NSConnectionReplyMode	Loops that operate while NSConnection is waiting for reply.

Events that are skipped are left in the queue.

You can use this method to short circuit normal event dispatching and get your own events. For example, you may want to do this in response to a mouse-down event in order to track the mouse while it's down. In this case, you would set *mask* to accept mouse-dragged or mouse-up events and use the NSEventTrackingRunLoopMode.

See also: – **postEvent:atStart:**, – **run**, – **runModalForWindow:**

orderFrontColorPanel:

– (void)**orderFrontColorPanel:**(id)*sender*

Brings up the color panel, an instance of NSColorPanel. If the NSColorPanel does not exist yet, it creates one. This method is typically invoked when the user chooses Colors from a menu.

orderFrontDataLinkPanel:

– (void)**orderFrontDataLinkPanel:**(id)*sender*

Brings up the data link panel, an instance of NSDataLinkPanel. If the NSDataLinkPanel does not exist yet, it creates one. This method is typically invoked when the user chooses an appropriate command from the application's menu. For example, the Edit application invokes this method when the user chooses Link Inspector from the Link menu.

postEvent:atStart:

– (void)**postEvent:**(NSEvent *)*anEvent* **atStart:**(BOOL)*flag*

Adds *anEvent* to the application's event queue. If *flag* is YES, the event is added to the front of the queue, otherwise the event is added to the back of the queue.

See also: – **currentEvent**, – **sendEvent:**

preventWindowOrdering

– (void)**preventWindowOrdering**

Suppresses the usual window ordering in handling the most recent mouse-down event. This method is only useful for mouse-down events when you want to prevent the window that receives the event from being ordered to the front.

registerServicesMenuSendTypes:returnTypes:

– (void)**registerServicesMenuSendTypes:**(NSArray *)*sendTypes* **returnTypes:** (NSArray *)*returnTypes*

Registers the pasteboard types that the application can send and receive in response to service requests. If the application has a Services menu, a menu item is added for each service provider that can accept one of the specified *sendTypes* or return one of the specified *returnTypes*. You should typically invoke this method at application start-up time or when an object that can use services is created. You can invoke it more than once; its purpose is to ensure that there is a menu item for every service that the application may use. The event-handling mechanism will dynamically enable the individual items to indicate which services are currently appropriate. All of the NSResponders in your application (typically NSViews) should register every possible type that they can send and receive by sending this message to NSApp.

See also: – **validRequestorForSendType:returnType:**, – **readSelectionFromPasteboard:** (NSServicesRequests protocol), – **writeSelectionToPasteboard:types:** (NSServicesRequests protocol)

removeWindowsItem:

– (void)**removeWindowsItem:**(NSWindow *)*aWindow*

Removes the Window menu item for *aWindow*. This method doesn't prevent the item from being automatically added again. Use NSWindow's **setExcludedFromWindowsMenu:** method if you want the item to remain excluded from the Window menu.

See also: – **addWindowsItem:title:filename:**, – **changeWindowsItem:title:filename:**

reportException:

– (void)**reportException:**(NSException *)*anException*

Logs *anException* by calling **NSLog()**. This method does not raise the exception. Use it inside of an exception handler to record that the exception occurred.

run

– (void)**run**

Starts the main event loop. The loop continues until a **stop:** or **terminate:** message is received. Upon each iteration through the loop, the next available event from the Window Server is stored and is then dispatched by sending the event to NSApp using **sendEvent:**.

Send a **run** message as the last statement from **main()**, after the application's objects have been initialized.

See also: – **applicationDidFinishLaunching:** (delegate method), – **runModalForWindow:**, – **runModalSession:**

runModalForWindow:

– (int)**runModalForWindow:**(NSWindow *)*aWindow*

Starts a modal event loop for *aWindow*. Until the loop is broken by a **stopModal**, **stopModalWithCode:**, or **abortModal** message, the application won't respond to any mouse, keyboard, or window-close events unless they're associated with *aWindow*. If **stopModalWithCode:** is used to stop the modal event loop, this method returns the argument passed to **stopModalWithCode:**. If **stopModal** is used, it returns the constant NSRunStoppedResponse. If **abortModal** is used, it returns the constant NSRunAbortedResponse. This method is functionally similar to the following:

```
NSModalSession session = [NSApp beginModalSessionForWindow:theWindow];

for (;;) {
    if ([NSApp runModalSession:session] != NSRunContinuesResponse)
        break;
}
[NSApp endModalSession:session];
```

The window *aWindow* is placed on the screen and made key as a result of the **runModalForWindow:** message. Do not send **makeKeyAndOrderFront:** to *aWindow*.

See also: – **run**, – **runModalSession:**

runModalSession:

– (int)**runModalSession:**(NSModalSession)*session*

Runs a modal session represented by *session*, as defined in a previous invocation of **beginModalSessionForWindow:**. A loop using this method is similar to a modal event loop run with **runModalForWindow:**, except that with this method the application can continue processing between method invocations. When you invoke this method, events for the NSWindow of this session are dispatched as normal; this method returns when there are no more events. You must invoke this method frequently enough that the window remains responsive to events.

If the modal session was not stopped, this method returns NSRunContinuesResponse. If **stopModal** was invoked as the result of event processing, NSRunStoppedResponse is returned. If **stopModalWithCode:** was invoked, this method returns the value passed to **stopModalWithCode:**. The NSAbortModalException raised by **abortModal** isn't caught, so **abortModal** will not stop the loop.

The window is placed on the screen and made key as a result of the **runModalSession:** message. Do not send a separate **makeKeyAndOrderFront:** message.

See also: – **endModalSession:**, – **run**

runPageLayout:

– (void)**runPageLayout:**(id)*sender*

Displays the application's page layout panel, an instance of NSPageLayout. If the NSPageLayout instance does not exist, it creates one. This method is typically invoked when the user selects Page Layout from the application's menu.

sendAction:to:from:

– (BOOL)**sendAction:**(SEL)*anAction* **to:**(id)*aTarget* **from:**(id)*sender*

Sends the message *anAction* to *aTarget*. If *aTarget is* **nil**, NSApp looks for an object that can respond to the message—that is, an object that implements a method matching *anAction*. It begins with the first responder of the key window. If the first responder can't respond, it tries the first responder's next responder and continues following next responder links up the responder chain. If none of the objects in the key window's responder chain can handle the message, NSApp attempts to send the message to the key window's delegate.

If the delegate doesn't respond and the main window is different from the key window, NSApp begins again with the first responder in the main window. If objects in the main window can't respond, NSApp attempts to send the message to the main window's delegate. If still no object has responded, NSApp tries to handle the message itself. If NSApp can't respond, it attempts to send the message to its own delegate.

Returns YES if the action is successfully sent; otherwise returns NO.

See also: – **targetForAction:**, – **tryToPerform:with:**, – **makeWindowsPerform:inOrder:**

sendEvent:

– (void)**sendEvent:**(NSEvent *)*anEvent*

Dispatches *anEvent* to other objects. You rarely invoke **sendEvent:** directly although you might want to override this method to perform some action on every event. **sendEvent:** messages are sent from the main event loop (the **run** method). **sendEvent:** is the method that dispatches events to the appropriate responders; NSApp handles application events, the NSWindow indicated in the event record handles window related events, and mouse and key events are forwarded to the appropriate NSWindow for further dispatching.

See also: – **currentEvent**, – **postEvent:atStart:**

servicesMenu

– (NSMenu *)**servicesMenu**

Returns the Services menu or **nil** if no Services menu has been created.

See also: – **setServicesMenu:**

servicesProvider

– (id)**servicesProvider**

Returns the object that provides the services that this application advertises in the Services menu of other applications.

See also: – **setServicesProvider:**

setApplicationIconImage:

– (void)**setApplicationIconImage:**(NSImage *)*anImage*

Sets the application's icon to *anImage.*

See also: – **applicationIconImage**

setDelegate:

– (void)**setDelegate:**(id)*anObject*

Makes *anObject* the receiver's delegate. The messages that a delegate can expect to receive are listed at the end of this specification. The delegate doesn't need to implement all the methods.

See also: – **delegate**

setMainMenu:

– (void)**setMainMenu:**(NSMenu *)*aMenu*

Makes *aMenu* the application's main menu.

See also: – **mainMenu**

setServicesMenu:

– (void)**setServicesMenu:**(NSMenu *)*aMenu*

Makes *aMenu* the application's Services menu.

See also: – **servicesMenu**

setServicesProvider:

– (void)**setServicesProvider:**(id)*aProvider*

Registers the object *aProvider* as the service provider. The service provider is an object that performs all of the services that the application provides to other applications. When another application requests a service from the receiver, it sends the service request to *aProvider*.

For more information on registering services, see the on-line document **/NextLibrary/Documentation/NextDev/TasksAndConcepts/ProgrammingTopics/Services.rtf**.

See also: – **servicesProvider**

setWindowsMenu:

– (void)**setWindowsMenu:**(NSMenu *)*aMenu*

Makes *aMenu* the application's Window menu.

See also: – **windowsMenu**

setWindowsNeedUpdate:

– (void)**setWindowsNeedUpdate:**(BOOL)*flag*

Sets whether the application's windows need updating when the application has finished processing the current event. This method is especially useful for making sure menus are updated to reflect changes not initiated by user actions, such as messages received from remote objects.

See also: – **updateWindows**

showHelp:

– (void)**showHelp:**(id)*sender*

Brings up the application's help file by sending a request to the shared NSWorkspace object to open the file using the default application for the help file's type. (You set the application's help file using Project Builder.) This method is typically invoked when the user chooses the Help command or one of the commands from the Help menu.

On Microsoft Windows platforms, the help file is typically an HLP file, so this method brings up Microsoft Windows help. On Mach platforms, the help file is typically an RTF file and is displayed using Edit, but the help file can be anything. For example, Project Builder on Mach brings up a Digital Librarian bookshelf in response to its Help command.

For more information on providing on-line help for your application, see the NSHelpManager class specification.

See also: – **activateContextHelpMode:**

stop:

– (void)**stop:**(id)*sender*

Stops the main event loop. This method will break the flow of control out of the **run** method, thereby returning to the **main()** function. A subsequent **run** message will restart the loop.

If this method is invoked during a modal event loop, it will break that loop but not the main event loop.

See also: – **runModalForWindow:**, – **runModalSession:**, – **terminate:**

stopModal

– (void)**stopModal**

Stops a modal event loop. This method should always be paired with a previous invocation of **runModalForWindow:** or **beginModalSessionForWindow:**. When **runModalForWindow:** is stopped with this method, it returns NSRunStoppedResponse. This method will stop the loop only if it's executed by code responding to an event. If you need to stop a **runModalForWindow:** loop from a method registered with the current NSRunLoop (for example, a method repeatedly invoked by an NSTimer object), use the **abortModal** method.

See also: – **runModalSession:**, – **stopModalWithCode:**

stopModalWithCode:

– (void)**stopModalWithCode:**(int)*returnCode*

Like **stopModal**, except the argument *returnCode* allows you to specify the value that **runModalForWindow:** will return.

See also: – **abortModal**

targetForAction:

– (id)**targetForAction:**(SEL)*aSelector*

Returns the object that receives the action message *aSelector.*

See also: – **sendAction:to:from:**, – **tryToPerform:with:**

terminate:

– (void)**terminate:**(id)*sender*

Terminates the application. This method is typically invoked when the user chooses Quit or Exit from the application's menu. Each use of **terminate:** invokes **applicationShouldTerminate:** to notify the delegate that the application will terminate. If **applicationShouldTerminate:** returns NO, control is returned to the main event loop, and the application isn't terminated. Otherwise, this method posts an NSApplicationWillTerminateNotification to the default notification center. Don't put final cleanup code in your application's **main()** function; it will never be executed. If cleanup is necessary, have the delegate respond to **applicationWillTerminate:** and perform the cleanup in that method.

See also: – **run**, – **stop:**

tryToPerform:with:

– (BOOL)**tryToPerform:**(SEL)*aSelector* **with:**(id)*anObject*

Dispatches action messages. The receiver tries to perform the method *aSelector* using its inherited NSResponder method **tryToPerform:with:**. If the receiver doesn't perform *aSelector*, the delegate is given the opportunity to perform it using its inherited NSObject method **performSelector:withObject:**. If either the receiver or its delegate accept *aSelector*, this method returns YES; otherwise it returns NO.

See also: – **respondsToSelector:** (NSObject)

unhide:

– (void)**unhide:**(id)*sender*

Restores hidden windows to the screen and makes the application active. Invokes **unhideWithoutActivation**.

See also: – **activateIgnoringOtherApps:**, – **hide:**

unhideWithoutActivation

– (void)**unhideWithoutActivation**

Restores hidden windows without activating their owner (the receiver). When this method begins, it posts an NSApplicationWillUnhideNotification to the default notification center. If it completes successfully, it posts an NSApplicationDidUnhideNotification.

See also: – **activateIgnoringOtherApps:**, – **applicationDidUnhide:** (delegate method), – **applicationWillUnhide:** (delegate method), – **hide:**

updateWindows

– (void)**updateWindows**

Sends an **update** message to each on-screen NSWindow. This method is invoked automatically in the main event loop after each event. If the NSWindow has automatic updating turned on, its **update** method will redraw all of the NSWindow's NSViews that need redrawing. If automatic updating is turned off, the **update** message does nothing. (You turn automatic updating on and off by sending **setAutodisplay:** to an NSWindow.)

When this method begins, it posts an NSApplicationWillUpdateNotification to the default notification center. When it successfully completes, it posts an NSApplicationDidUpdateNotification.

See also: – **applicationWillUpdate:** (delegate method), – **applicationDidUpdate:** (delegate method), – **setWindowsNeedUpdate:**, – **setAutodisplay:** (NSWindow)

updateWindowsItem:

– (void)**updateWindowsItem:**(NSWindow *)*aWindow*

Updates the Window menu item for *aWindow* to reflect the edited status of *aWindow*. You rarely need to invoke this method because it is invoked automatically when the edit status of an NSWindow is set.

See also: – **changeWindowsItem:title:filename:**, – **setDocumentEdited:** (NSWindow)

validRequestorForSendType:returnType:

– (id)**validRequestorForSendType:**(NSString *)*sendType* **returnType:**(NSString *)*returnType*

Indicates whether the receiver can send and receive the specified pasteboard types. This message is sent to all responders in a responder chain. NSApp is typically the last item in the responder chain, so it usually only receives this message if none of the current responders can send *sendType* data and accept back *returnType* data.

The receiver passes this message on to its delegate if the delegate can respond (and isn't an NSResponder with its own next responder). If the delegate can't respond or returns **nil**, this method returns **nil**. If the delegate can find an object that can send *sendType* data and accept back *returnType* data, that object is returned.

See also: – **validRequestorForSendType:returnType:** (NSResponder), – **registerServicesMenuSendTypes:returnTypes:**, – **readSelectionFromPasteboard:** (NSServicesRequests protocol), – **writeSelectionToPasteboard:types:** (NSServicesRequests protocol)

windows

– (NSArray *)**windows**

Returns an NSArray of the application's NSWindows, including off-screen windows.

windowsMenu

– (NSMenu *)**windowsMenu**

Returns the Window menu or **nil** if no Window menu has been created.

See also: – **setWindowsMenu:**

windowWithWindowHandle:

– (NSWindow *)**windowWithWindowHandle:**(void *)*hWnd*

On Microsoft Windows platforms, returns the NSWindow object associated with the Win32 window handle *hWnd*. If the application does not own *hWnd* or *hWnd* does not have an NSWindow associated with it, this method returns **nil**. This method is for Microsoft Windows platforms only. **windowWithWindowHandle:** is not defined for the Mach platform.

See also: – **windowHandle** (NSWindow)

windowWithWindowNumber:

– (NSWindow *)**windowWithWindowNumber:**(int)*windowNum*

Returns the NSWindow object corresponding to *windowNum*.

Notifications

NSApplicationDidBecomeActiveNotification

Posted immediately after the application becomes active. This notification contains a notification object but no userInfo dictionary. The notification object is NSApp.

NSApplicationDidFinishLaunchingNotification

Posted at the end of the **finishLaunching** method to indicate that the application has completed launching and is ready to run. This notification contains a notification object but no userInfo dictionary. The notification object is NSApp.

NSApplicationDidHideNotification

Posted at the end of the **hide:** method to indicate that the application is now hidden. This notification contains a notification object but no userInfo dictionary. The notification object is NSApp.

NSApplicationDidResignActiveNotification

Posted immediately after the application gives up its active status to another application. This notification contains a notification object but no userInfo dictionary. The notification object is NSApp.

NSApplicationDidUnhideNotification

Posted at the end of the **unhideWithoutActivation** method to indicate that the application is now visible. This notification contains a notification object but no userInfo dictionary. The notification object is NSApp.

NSApplicationDidUpdateNotification

Posted at the end of the **updateWindows** method to indicate that the application has finished updating its windows. This notification contains a notification object but no userInfo dictionary. The notification object is NSApp.

NSApplicationWillBecomeActiveNotification

Posted immediately after the application becomes active. This notification contains a notification object but no userInfo dictionary. The notification object is NSApp.

NSApplicationWillFinishLaunchingNotification

Posted at the start of the **finishLaunching** method to indicate that the application has completed its initialization process and is about to finish launching. This notification contains a notification object but no userInfo dictionary. The notification object is NSApp.

NSApplicationWillHideNotification

Posted at the start of the **hide:** method to indicate that the application is about to be hidden. This notification contains a notification object but no userInfo dictionary. The notification object is NSApp.

NSApplicationWillResignActiveNotification

Posted immediately before the application gives up its active status to another application. This notification contains a notification object but no userInfo dictionary. The notification object is NSApp.

NSApplicationWillTerminateNotification

Posted by the **terminate:** method to indicate that the application will terminate. Posted only if the delegate method **applicationShouldTerminate:** returns YES. This notification contains a notification object but no userInfo dictionary. The notification object is NSApp.

NSApplicationWillUnhideNotification

Posted at the start of the **unhideWithoutActivation** method to indicate that the application is about to be visible. This notification contains a notification object but no userInfo dictionary. The notification object is NSApp.

NSApplicationWillUpdateNotification

Posted at the start of the **updateWindows** method to indicate that the application is about to update its windows. This notification contains a notification object but no userInfo dictionary. The notification object is NSApp.

Methods Implemented By the Delegate

application:openFile:

– (BOOL)**application:**(NSApplication *)*theApplication* **openFile:**(NSString *)*filename*

Sent directly by *theApplication* to the delegate. The method should open the file *filename*, returning YES if the file is successfully opened, and NO otherwise.

Note: If the user has started up the application by double-clicking a file, the delegate receives the **application:openFile:** message before receiving **applicationDidFinishLaunching:**. (**applicationWillFinishLaunching:** is sent before **application:openFile:**.)

See also: – **application:openFileWithoutUI:**, – **application:openTempFile:**, – **applicationOpenUntitledFile:**

application:openFileWithoutUI:

– (BOOL)**application:**(NSApplication *)*sender* **openFileWithoutUI:**(NSString *)*filename*

Sent directly by *sender* to the delegate to request that the file *filename* be opened as a linked file. The method should open the file without bringing up its application's user interface; that is, work with the file is under programmatic control of *sender*, rather than under keyboard control of the user. Returns YES if the file was successfully opened, NO otherwise.

See also: – **application:openFile:**, – **application:openTempFile:**, – **application:printFile:**, – **applicationOpenUntitledFile:**

application:openTempFile:

– (BOOL)**application:**(NSApplication *)*theApplication* **openTempFile:**(NSString *)*filename*

Sent directly by *theApplication* to the delegate. The method should attempt to open the file *filename*, returning YES if the file is successfully opened, and NO otherwise.

By design, a file opened through this method is assumed to be temporary; it's the application's responsibility to remove the file at the appropriate time.

See also: – **application:openFile:**, – **application:openFileWithoutUI:**, – **applicationOpenUntitledFile:**

application:printFile:

– (BOOL)**application:**(NSApplication *)*theApplication* **printFile:**(NSString *)*filename*

Sent when the user starts up the application on the command line with the **-NSPrint** option. Sent directly by *theApplication* to the delegate.

The method should attempt to print the file *filename*, returning YES if the file was successfully printed, and NO otherwise. The application terminates (using the **terminate:** method) after this method returns.

If at all possible, this method should print the file without displaying the user interface. For example, if you pass the **-NSPrint** option to the TextEdit application, TextEdit assumes you want to print the entire contents of the specified file. However, if the application opens more complex documents, you may want to display a panel that lets user choose exactly what they want to print.

See also: – **application:openFileWithoutUI:**

applicationDidBecomeActive:

– (void)**applicationDidBecomeActive:**(NSNotification *)*aNotification*

Sent by the default notification center immediately after the application becomes active. *aNotification* is always an NSApplicationDidBecomeActiveNotification. You can retrieve the NSApplication object by sending the **object** method to *aNotification*.

See also: – **applicationDidFinishLaunching:**, – **applicationDidResignActive:**, – **applicationWillBecomeActive:**

applicationDidFinishLaunching:

– (void)**applicationDidFinishLaunching:**(NSNotification *)*aNotification*

Sent by the default notification center after the application has been launched and initialized but before it has received its first event. *aNotification* is always an NSApplicationDidFinishLaunchingNotification. You can retrieve the NSApplication object in question by sending **object** to *aNotification*. The delegate can implement this method to perform further initialization.

Note: If the user has started up the application by double-clicking a file, the delegate receives the **application:openFile:** message before receiving **applicationDidFinishLaunching:**. (**applicationWillFinishLaunching:** is sent before **application:openFile:**.)

See also: – **applicationDidBecomeActive:**, – **finishLaunching** (NSApplication class)

applicationDidHide:

– (void)**applicationDidHide:**(NSNotification *)*aNotification*

Sent by the default notification center immediately after the application is hidden. *aNotification* is always an NSApplicationDidHideNotification. You can retrieve the NSApplication object in question by sending **object** to *aNotification*.

See also: – **applicationWillHide:**, – **applicationDidUnhide:**, – **hide**: (NSApplication class)

applicationDidResignActive:

– (void)**applicationDidResignActive:**(NSNotification *)*aNotification*

Sent by the default notification center immediately after the application is deactivated. *aNotification* is always an NSApplicationDidResignActiveNotification. You can retrieve the NSApplication object in question by sending **object** to *aNotification*.

See also: – **applicationDidBecomeActive:**, – **applicationWillResignActive:**

applicationDidUnhide:

– (void)**applicationDidUnhide:**(NSNotification *)*aNotification*

Sent by the default notification center immediately after the application is made visible. *aNotification* is always an NSApplicationDidUnhideNotification. You can retrieve the NSApplication object in question by sending **object** to *aNotification*.

See also: – **applicationDidHide:**, – **applicationWillUnhide:**, – **unhide**: (NSApplication class)

applicationDidUpdate:

– (void)**applicationDidUpdate:**(NSNotification *)*aNotification*

Sent by the default notification center immediately after the NSApplication object updates its NSWindows. *aNotification* is always an NSApplicationDidUpdateNotification. You can retrieve the NSApplication object in question by sending **object** to *aNotification*.

See also: – **applicationWillUpdate:**, – **updateWindows** (NSApplication class)

applicationOpenUntitledFile:

– (BOOL)**applicationOpenUntitledFile:**(NSApplication *)*theApplication*

Sent directly by *theApplication* to the delegate to request that a new, untitled file be opened. Returns YES if the file was successfully opened, NO otherwise.

See also: – **application:openFile:**, – **application:openFileWithoutUI:**, – **application:openTempFile:**

applicationShouldTerminate:

– (BOOL)**applicationShouldTerminate:**(NSApplication *)*sender*

Invoked from within the **terminate:** method immediately before the application terminates. *sender* is the NSApplication to be terminated. If this method returns NO, the application is not terminated, and control returns to the main event loop. Return YES to allow the application to terminate.

See also: – **applicationShouldTerminateAfterLastWindowClosed:**, – **applicationWillTerminate:**, – **terminate:** (NSApplication class)

applicationShouldTerminateAfterLastWindowClosed:

– (BOOL)**applicationShouldTerminateAfterLastWindowClosed:**(NSApplication *)*theApplication*

Invoked when the user closes the last window that the application has open on.

This method is intended for the Microsoft Windows platform. On Microsoft Windows, the default behavior is to terminate the application if the user closes the last window. Most application use this default behavior; however, you may choose to have **applicationShouldTerminateAfterLastWindowClosed:** perform some other function, such as display a panel that gives the user a choice of exiting the application or opening a new window.

If this method returns NO, the application is not terminated, and control returns to the main event loop. Return YES to allow the application to terminate. Note that **applicationShouldTerminate:** is invoked if this method returns YES.

See also: – **terminate:** (NSApplication class)

applicationWillBecomeActive:

– (void)**applicationWillBecomeActive:**(NSNotification *)*aNotification*

Sent by the default notification center immediately before the application becomes active. *aNotification* is always an NSApplicationWillBecomeActiveNotification. You can retrieve the NSApplication object in question by sending **object** to *aNotification*.

See also: – **applicationDidBecomeActive:**, – **applicationWillFinishLaunching:**, – **applicationWillResignActive:**

applicationWillFinishLaunching:

– (void)**applicationWillFinishLaunching:**(NSNotification *)*aNotification*

Sent by the default notification center immediately before the NSApplication object is initialized. *aNotification* is always an NSApplicationWillFinishLaunchingNotification. You can retrieve the NSApplication object in question by sending **object** to *aNotification*.

See also: – **applicationDidFinishLaunching:**, – **applicationWillBecomeActive:**, – **finishLaunching** (NSApplication class)

applicationWillHide:

– (void)**applicationWillHide:**(NSNotification *)*aNotification*

Sent by the default notification center immediately after the application is hidden. *aNotification* is always an NSApplicationWillHideNotification. You can retrieve the NSApplication object in question by sending **object** to *aNotification*.

See also: – **applicationDidHide:**, – **hide**: (NSApplication class)

applicationWillTerminate:

– (void)**applicationWillTerminate:**(NSNotification *)*aNotification*

Sent by the default notification center immediately before the application terminates. *aNotification* is always an NSApplicationWillTerminateNotification. You can retrieve the NSApplication object in question by sending **object** to *aNotification*. Put any necessary cleanup code in this method.

See also: – **applicationShouldTerminate:**, – **terminate:** (NSApplication class)

applicationWillResignActive:

– (void)**applicationWillResignActive:**(NSNotification *)*aNotification*

Sent by the default notification center immediately after the application is deactivated. *aNotification* is always an NSApplicationWillResignActiveNotification. You can retrieve the NSApplication object in question by sending **object** to *aNotification*.

See also: – **applicationWillBecomeActive:**, – **applicationDidResignActive:**

applicationWillUnhide:

– (void)**applicationWillUnhide:**(NSNotification *)*aNotification*

Sent by the default notification center immediately after the application is unhidden. *aNotification* is always an NSApplicationWillUnhideNotification. You can retrieve the NSApplication object in question by sending **object** to *aNotification*.

See also: – **applicationDidUnhide:**, – **applicationWillHide:**, – **unhide**: (NSApplication class)

applicationWillUpdate:

– (void)**applicationWillUpdate:**(NSNotification *)*aNotification*

Sent by the default notification center immediately before the NSApplication object updates its NSWindows. *aNotification* is always an NSApplicationWillUpdateNotification. You can retrieve the NSApplication object in question by sending **object** to *aNotification*.

See also: – **applicationDidUpdate:**, – **updateWindows** (NSApplication class)

NSAttributedString Class Cluster Additions

Class Cluster Description

NSAttributedString objects manage character strings and associated sets of attributes (for example, font and kerning) that apply to individual characters or ranges of characters in the string. An association of characters and their attributes is called an *attributed string*. The cluster's two public classes, NSAttributedString and NSMutableAttributedString, declare the programmatic interface for read-only attributed strings and modifiable attributed strings, respectively. The Foundation Kit defines the basic functionality for attributed strings, while the remainder is defined here in the Application Kit. The Application Kit also uses a subclass of NSMutableAttributedString, called NSTextStorage, to provide the storage for the extended text-handling system.

Note: NSAttributedString is *not* a subclass of NSString. It contains a string object to which it applies attributes. This protects users of attributed strings from ambiguities caused by the semantic differences between simple and attributed string. For example, equality can't be simply defined between an NSString and an attributed string.

Because of the nature of class clusters, attributed string objects are not actual instances of the NSAttributedString or NSMutableAttributedString classes, but are instances of one of their private concrete subclasses. Although an attributed string object's class is private, its interface is public, as declared by these abstract superclasses, NSAttributedString and NSMutableAttributedString. The attributed string classes adopt the NSCopying and NSMutableCopying protocols, making it convenient to convert an attributed string from one type to the other.

You create an NSAttributedString object from scratch by using one of **initWithString:**, **initWithString: attributes:**, or **initWithAttributedString:**. These methods initialize an attributed string with data you provide. You can also create an attributed string from RTF data using **initWithRTF:documentAttributes:** , **initWithRTFD:documentAttributes:**, **initWithRTFDFileWrapper:documentAttributes:**, or **initWithPath:documentAttributes:**. See "RTF Document Attributes" for more details on reading RTF.

An attributed string provides basic access to its contents with the **string** and **attributesAtIndex: effectiveRange:** methods, which yield characters and attributes, respectively. These two primitive methods are used by the other access methods to retrieve attributes individually by name, by functional group (font or ruler attributes, for example), and so on.

Feature Overview

NSAttributedString and NSMutableAttributedString add a number of features to the basic content storage of NSString:

- Association of arbitrary, programmer-defined attributes with ranges of characters
- Preservation of attribute-to-character mapping after changes (NSMutableAttributedString)

- Support for RTF, including file attachments and graphics
- Drawing in NSView objects (note that the Application Kit adds drawing methods to NSString as well)
- Linguistic unit (word) and line calculation

An attributed string identifies attributes by name, storing their values as opaque **id**s in an NSDictionary. For example, the text font is stored as an NSFont object under the name given by NSFontAttributeName. You can associate any object value, by any name, with a given range of characters in the attributed string. The basic attributes defined by NSAttributedString are described under "Accessing Attributes" below.

A mutable attributed string keeps track of the attribute mapping as characters are added to and deleted from it and as attributes are changed. It allows you to group batches of edits with the **beginEditing** and **endEditing** methods, and to consolidate changes to the attribute-to-character mapping with the **fix...** methods. See "Changing a Mutable Attributed String" below for more information.

An attributed string can be created from rich text (RTF) or rich text with attachments (RTFD), and can write its contents as RTF or RTFD data. Three initialization methods, **initWithRTF:documentAttributes:**, **initWithRTFD:documentAttributes:**, and **initWithRTFDFileWrapper:documentAttributes:**, interpret rich text data. The methods for writing rich text are **RTFFromRange:documentAttributes:** and **RTFDFromRange:documentAttributes:**, which return rich text data for any legal range within the attributed string, and **RTFDFileWrapperFromRange:documentAttributes:**, which returns the attributed string as an NSFileWrapper. NSAttributedString provides limited support for some document-level attributes, as described under "RTF Document Attributes" below. Additional support for rich text is provided by other text-handling classes such as NSTextView.

You can draw an attributed string in a focused NSView by invoking either the **drawAtPoint:** or **drawInRect:** method. Note that the Application Kit defines drawing methods for NSString as well, allowing any string object to draw itself. These methods, **drawAtPoint:withAttributes:** and **drawInRect: withAttributes:**, are described in the NSString Additions class specification.

An attributed string supports the typical behavior of editors in selecting a word on a double-click with the **doubleClickAtIndex:** method, and finds word breaks with **nextWordFromIndex:forward:**. It also calculates line breaks with the **lineBreakBeforeIndex:withinRange:** method.

Accessing Attributes

An attributed string identifies attributes by name, storing an **id** value under the attribute name in an NSDictionary, which is in turn associated with an NSRange that indicates the characters to which the dictionary's attributes apply. You can assign any attribute name/value pair you wish to a range of characters, in addition to these standard attributes:

Attribute Identifier	Value Class	Default Value
NSFontAttributeName	NSFont	Helvetica 12-point
NSForegroundColorAttributeName	NSColor	black

Attribute Identifier	Value Class	Default Value
NSBackgroundColorAttributeName	NSColor	none (no background drawn)
NSUnderlineStyleAttributeName	NSNumber, as an **int**	none (no underline)
NSSuperscriptAttributeName	NSNumber, as an **int**	0
NSBaselineOffsetAttributeName	NSNumber, as a **float**	0.0
NSKernAttributeName	NSNumber, as a **float**	0.0
NSLigatureAttributeName	NSNumber, as an **int**	1 (standard ligatures)
NSParagraphStyleAttributeName	NSParagraphStyle	(as returned by NSParagraphStyle's **defaultParagraphStyle** method)
NSAttachmentAttributeName	NSTextAttachment	none (no attachment)

The identifiers listed are actually global NSString variables containing the attribute names. The value class is what users of an attributed string should expect the attribute values to be presented as. The default values are what they should assume if no attribute value has been explicitly set for the requested character range.

The natures of several attributes aren't obvious from name alone:

- The underline attribute has only one value defined, NSSingleUnderlineStyle. All characters with this attribute value should be drawn with a single line just below the baseline.
- The superscript attribute indicates an abstract level for both super- and subscripts. The user of the attributed string can interpret this as desired, adjusting the baseline by the same or a different amount for each level, changing the font size, or both.
- The baseline offset attribute is a literal distance by which the characters should be shifted above the baseline (for positive offsets) or below (for negative offsets).
- The kerning attribute indicates how much the following character should be shifted from its default offset as defined by the current character's font; a positive kern indicates a shift farther along and a negative kern indicates a shift closer to the current character.
- The ligature attribute determines what kinds of ligatures should be used when displaying the string. A value of 0 indicates that only ligatures essential for proper rendering of text should be used, 1 indicates that standard ligatures should be used, and 2 indicates that all available ligatures should be used. Which ligatures are standard depends on the script and possibly the font. Arabic text, for example, requires ligatures for many character sequences, but has a rich set of additional ligatures that combine characters. English text has no essential ligatures, and typically has only two standard ligatures, those for "fi" and "fl"—all others being considered more advanced or fancy.

With an immutable attributed string, you assign all attributes on creating the string using methods such as **initWithRTF:documentAttributes:**, which interprets attributes from the RTF data, **initWithString: attributes:**, which explicitly takes an NSDictionary of name/value pairs, or **initWithString:**, which assigns no attributes. See “Changing a Mutable Attributed String” below for information on assigning attributes with a mutable attributed string.

To retrieve attribute values from either type of attributed string, use any of these methods:

– attributesAtIndex:effectiveRange:
– attributesAtIndex:longestEffectiveRange:inRange:
– attribute:atIndex:effectiveRange:
– attribute:atIndex:longestEffectiveRange:inRange:
– fontAttributesInRange:
– rulerAttributesInRange:

The first two methods return all attributes at a given index, the **attribute:...** methods return the value of a single named attribute, and **fontAttributesInRange:** and **rulerAttributesInRange:** return attributes defined to apply only to characters or to whole paragraphs, respectively (see the individual method descriptions for more information).

The first four methods also return by reference the *effective range* and the *longest effective range* of the attributes. These ranges allow you to determine the extent of attributes. Conceptually, each character in an attributed string has its own collection of attributes; however, it’s often useful to know when the attributes and values are the same over a series of characters. This allows a routine to progress through an attributed string in chunks larger than a single character. In retrieving the effective range, an attributed string simply looks up information in its attribute mapping, essentially the dictionary of attributes that apply at the index requested. In retrieving the longest effective range, the attributed string continues checking characters past this basic range as long as the attribute values are the same. This extra comparison increases the execution time for these methods but guarantees a precise maximal range for the attributes requested.The code fragment below progresses through an attributed string in chunks based on the effective range. The fictitious **analyzer** object here counts the number of characters in each font. The **while** loop progresses as long as the effective range retrieved doesn’t include the end of the attributed string, retrieving the font in effect just past the latest retrieved range. For each font attribute retrieved, **analyzer** is asked to tally the number of characters in the effective range. In this example, it’s possible that consecutive invocations of **attribute: atIndex:effectiveRange:** will return the same value.

```
NSAttributedString *attrStr;
unsigned int length;
NSRange effectiveRange;
id attributeValue;

length = [attrStr length];
effectiveRange = NSMakeRange(0, 0);

while (NSMaxRange(effectiveRange) < length) {
    attributeValue = [attrStr attribute:NSFontAttributeName
        atIndex:NSMaxRange(effectiveRange) effectiveRange:&effectiveRange];
    [analyzer tallyCharacterRange:effectiveRange font:attributeValue];
}
```

In contrast, the next code fragment progresses through the attributed string according to the maximum effective range for each font. In this case, **analyzer** counts font *changes*, which may not be represented by merely retrieving effective ranges. In this case the **while** loop is predicated on the length of the limiting range, which begins as the entire length of the attributed string and is whittled down as the loop progresses. After **analyzer** records the font change, the limit range is adjusted to account for the longest effective range retrieved.

```
NSAttributedString *attrStr;
NSRange limitRange;
NSRange effectiveRange;
id attributeValue;

limitRange = NSMakeRange(0, [attrStr length]);

while (limitRange.length > 0) {
    attributeValue = [attrStr attribute:NSFontAttributeName
        atIndex:limitRange.location longestEffectiveRange:&effectiveRange
        inRange:limitRange];
    [analyzer recordFontChange:attributeValue];
    limitRange = NSMakeRange(NSMaxRange(effectiveRange),
        NSMaxRange(limitRange) - NSMaxRange(effectiveRange));
}
```

Note that the second code fragment is more complex. Because of this, and because **attribute:atIndex: longestEffectiveRange:inRange:** is somewhat slower than **attribute:atIndex:effectiveRange:**, you should typically use it only when absolutely necessary for the work you're performing. In most cases working by effective range is enough.

Changing a Mutable Attributed String

NSMutableAttributedString declares a number of methods for changing both characters and attributes, such as the primitive **replaceCharactersInRange:withString:** and **setAttributes:range:**, or the more convenient methods **addAttribute:value:range:**, **applyFontTraits:range:**, **setAlignment:range:**, and so

on. All of the methods for changing a mutable attributed string properly update the mapping between characters and attributes, but after a change some inconsistencies can develop. Here are some examples of attribute consistency requirements:

- Paragraph styles must apply to entire paragraphs.
- Scripts may only be assigned fonts that support them. For example, Kanji and Arabic characters can't be assigned the Times-Roman font, and must be reassigned fonts that support these scripts.
- Deleting attachment characters from the string requires the corresponding attachment objects to be released. Similarly, removing attachment objects requires the corresponding attachment characters to be removed from the string.
- A code editing application that displays all language keywords in boldface can automatically assign this attribute as the user changes the font or edits the text.

NSMutableAttributedString defines methods to fix these inconsistencies as changes are made. This allows the attributes to be cleaned up at a low level, hiding potential problems from higher levels and providing for very clean update of display as attributes change. There are six methods for fixing attributes:

– fixAttributesInRange:
– fixAttachmentAttributeInRange:
– fixFontAttributeInRange:
– fixParagraphStyleAttributeInRange:
– beginEditing
– endEditing

The first method, **fixAttributesInRange:**, invokes the other three **fix...** methods to clean up deleted attachment references, font attributes, and paragraph attributes, respectively. The individual method descriptions explain what cleanup entails for each case.

The **beginEditing** and **endEditing** methods are provided for subclasses of NSMutableAttributedString to override. Their default implementations do nothing. These methods allow instances of a subclass to record or buffer groups of changes and clean themselves up on receiving an **endEditing** message. **endEditing** also allows the receiver to notify any observers that it has been changed. NSTextStorage's implementation of **endEditing**, for example, fixes changed attributes and then notifies its NSLayoutManagers that they need to re-lay and redisplay their text.

RTF Document Attributes

Attributed strings keep attribute information for their text only, while RTF allows for more general attributes of a document, especially regarding paper size and layout. To support higher-level objects that use attributed

strings, the methods that work with RTF also read and write some RTF directives for document attributes, stored in an NSDictionary under these keys:

Attribute Key	Value Class
PaperSize	NSValue, as an NSSize
LeftMargin	NSNumber, as a **float**
RightMargin	NSNumber, as a **float**
TopMargin	NSNumber, as a **float**
BottomMargin	NSNumber, as a **float**

The **init** methods, such as **initWithRTF:documentAttributes:**, return by reference a dictionary containing the attributes read from the RTF data, which your application can then use to set up its page layout. Similarly, RTF extraction methods such as **RTFFromRange:documentAttributes:**, accept a dictionary containing those attributes and writes them into the RTF data, thus preserving the page layout information.

Attachments

Attachments, such as embedded images or files, are represented in an attributed string by both a special character and an attribute. The character is identified by the global name NSAttachmentCharacter, and indicates the presence of an attachment at its location in the string. The attribute, identified in the string by the attribute name NSAttachmentAttributeName, is an NSTextAttachment object. An NSTextAttachment contains the data for the attachment itself, as well as an image to display when the string is drawn. You can use NSAttributedString's **attributedStringWithAttachment:** class method to construct an attachment string, which you can then add to a mutable attributed string using **appendAttributedString:** or **insertAttributedString:atIndex:**.

NSAttributedString Additions

Inherits From: NSObject

Declared In: AppKit/NSAttributedString.h
AppKit/NSStringDrawing.h
AppKit/NSTextAttachment.h

Class Description

The Application Kit extends the Foundation Kit's NSAttributedString class by adding:

- Support for RTF, with or without attachments
- Graphic attributes, including font and ruler attributes
- Methods for drawing attributed strings
- Methods for calculating significant linguistic units

Method Types

Creating an NSAttributedString

– initWithRTF:documentAttributes:
– initWithRTFD:documentAttributes:
– initWithRTFDFileWrapper:documentAttributes:
– initWithPath:documentAttributes:
– initWithHTML:documentAttributes:
+ attributedStringWithAttachment:

Retrieving attribute information

– fontAttributesInRange:
– rulerAttributesInRange:
– containsAttachments

Calculating linguistic units

– doubleClickAtIndex:
– lineBreakBeforeIndex:withinRange:
– nextWordFromIndex:forward:

Drawing the string

– drawAtPoint:
– drawInRect:
– size

Generating RTF data

– RTFFromRange:documentAttributes:
– RTFDFileWrapperFromRange:documentAttributes:
– RTFDFromRange:documentAttributes:

Class Methods

attributedStringWithAttachment:

+ (NSAttributedString *)**attributedStringWithAttachment:**(NSTextAttachment *)*attachment*

Returns an NSAttributedString object containing only the attachment marker character (NSAttachmentCharacter), which is assigned an attribute whose name is NSTextAttachmentName and whose value is *attachment*. Use this method, along with **appendAttributedString:** or **insertAttributedString:atIndex:**, to add an attachment to an attributed string.

Instance Methods

containsAttachments

– (BOOL)**containsAttachments**

Returns YES if the receiver contains any attachment attributes, NO otherwise. This method checks only for attachment attributes, not for NSAttachmentCharacter.

doubleClickAtIndex:

– (NSRange)**doubleClickAtIndex:**(unsigned)*index*

Returns the range of characters that form a word (or other linguistic unit) surrounding *index*, taking language characteristics into account. Raises an NSRangeException if *index* lies beyond the end of the receiver's characters.

See also: – **nextWordFromIndex:forward:**

drawAtPoint:

– (void)**drawAtPoint:**(NSPoint)*point*

Draws the receiver with its font and other display attributes at *point* in the currently focused NSView. Text is drawn in such a way that the upper left corner of its bounding box lies at *point*, regardless of the line sweep direction or whether the NSView is flipped.

Don't invoke this method while no NSView is focused.

See also: – **lockFocus** (NSView), – **size**, – **drawInRect:**

drawInRect:

– (void)**drawInRect:**(NSRect)*rect*

Draws the receiver with its font and other display attributes within *rect* in the currently focused NSView, clipping the drawing to this rectangle. Text is drawn within *rect* according to its line sweep direction; for example, Arabic text will begin at the right edge and potentially be clipped on the left.

Don't invoke this method while no NSView is focused.

See also: – **lockFocus** (NSView), – **drawAtPoint:**

fontAttributesInRange:

– (NSDictionary *)**fontAttributesInRange:**(NSRange)*aRange*

Returns the font attributes in effect for the character at *aRange.location*. Font attributes are all those listed under "Accessing Attributes" in the class cluster description except NSParagraphStyleAttributeName and NSAttachmentAttributeName. Use this method to obtain font attributes that are to be copied or pasted with "copy font" operations. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

See also: – **rulerAttributesInRange:**

initWithHTML:documentAttributes:

– (id)**initWithPath:**(NSData *)*data* **documentAttributes:**(NSDictionary **)*docAttributes*

Initializes and returns a new NSAttributedString from HTML contained in the object *data*. Also returns by reference in *docAttributes* a dictionary containing document-level attributes:

Key

Title
BaseURL
BackgroundImageURL
BackgroundColor
TextColor
LinkColor
ActiveLinkColor
VisitedLinkColor
LeftMargin
TopMargin

The parameter *docAttributes* can be NULL, in which case no document attributes are returned. Returns **nil** if the file at *path* can't be decoded.

initWithPath:documentAttributes:

– (id)**initWithPath:**(NSString *)*path* **documentAttributes:**(NSDictionary **)*docAttributes*

Initializes a new NSAttributedString from RTF or RTFD data contained in the file at *path*. Also returns by reference in *docAttributes* a dictionary containing document-level attributes, as listed in the class cluster description under the "RTF Document Attributes" heading. *docAttributes* may be NULL, in which case no document attributes are returned. Returns **self**, or **nil** if the file at *path* can't be decoded.

initWithRTF:documentAttributes:

– (id)**initWithRTF:**(NSData *)*rtfData* **documentAttributes:**(NSDictionary **)*docAttributes*

Initializes a new NSAttributedString by decoding the stream of RTF commands and data contained in *rtfData*. Also returns by reference in *docAttributes* a dictionary containing document-level attributes, as listed in the class cluster description under the "RTF Document Attributes" heading. *docAttributes* may be NULL, in which case no document attributes are returned. Returns **self**, or **nil** if *rtfData* can't be decoded.

initWithRTFD:documentAttributes:

– (id)**initWithRTFD:**(NSData *)*rtfdData* **documentAttributes:**(NSDictionary **)*docAttributes*

Initializes a new NSAttributedString by decoding the stream of RTFD commands and data contained in *rtfdData*. Also returns by reference in *docAttributes* a dictionary containing document-level attributes, as listed in the class cluster description under the "RTF Document Attributes" heading. *docAttributes* may be NULL, in which case no document attributes are returned. Returns **self**, or **nil** if *rtfData* can't be decoded.

initWithRTFDFileWrapper:documentAttributes:

– (id)**initWithRTFDFileWrapper:**(NSFileWrapper *)*wrapper* **documentAttributes:** (NSDictionary **)*docAttributes*

Initializes a new NSAttributedString from *wrapper* an NSFileWrapper object containing an RTFD document. Also returns by reference in *docAttributes* a dictionary containing document-level attributes, as listed in the class cluster description under the ““RTF Document Attributes” heading. *docAttributes* may be NULL, in which case no document attributes are returned. Returns **self**, or **nil** if *wrapper* can't be interpreted as an RTFD document.

lineBreakBeforeIndex:withinRange:

– (unsigned)**lineBreakBeforeIndex:**(unsigned)*index* **withinRange:**(NSRange)*aRange*

Returns the index of the closest character before *index* and within *aRange* that can be placed on a new line when laying out text. In other words, finds the appropriate line break when the character at *index* won't fit on the same line as the character at the beginning of *aRange*. Returns NSNotFound if no line break is possible before *index*.

Raises an NSRangeException if *index* or any part of *aRange* lies beyond the end of the receiver's characters.

See also: – **nextWordFromIndex:forward:**

nextWordFromIndex:forward:

– (unsigned)**nextWordFromIndex:**(unsigned)*index* **forward:**(BOOL)*flag*

Returns the index of the first character of the word after or before *index*. If *flag* is YES, this is the first character after *index* that begins a word; if *flag* is NO, it's the first character before *index* that begins a word, whether *index* is located within a word or not. If *index* lies at either end of the string and the search direction would progress past that end, it's returned unchanged. This method is intended for moving the insertion point during editing, not for linguistic analysis or parsing of text.

Raises an NSRangeException if *index* lies beyond the end of the receiver's characters.

See also: – **lineBreakBeforeIndex:withinRange:**

RTFDFileWrapperFromRange:documentAttributes:

– (NSFileWrapper *)**RTFDFileWrapperFromRange:**(NSRange)*aRange* **documentAttributes:** (NSDictionary *)*docAttributes*

Returns an NSFileWrapper object that contains an RTFD document corresponding to the characters and attributes within *aRange*. The file wrapper also includes the document-level attributes in *docAttributes*, as explained in the class cluster description under the “RTF Document Attributes” heading. If there are no

document-level attributes, *docAttributes* can be **nil**. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

You can save the file wrapper using NSFileWrapper's **writeToFile:atomically:updateFilenames:** method.

See also: – **RTFDFromRange:documentAttributes:**, – **RTFFromRange:documentAttributes:**

RTFDFromRange:documentAttributes:

– (NSData *)**RTFDFromRange:**(NSRange)*aRange* **documentAttributes:** (NSDictionary *)*docAttributes*

Returns an NSData object that contains an RTFD stream corresponding to the characters and attributes within *aRange*. Also writes the document-level attributes in *docAttributes*, as explained in the class cluster description under the "RTF Document Attributes" heading. If there are no document-level attributes, *docAttributes* can be **nil**. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

When writing data to the pasteboard, you can use the NSData object as the first argument to NSPasteboard's **setData:forType:** method, with a second argument of NSRTFDPboardType.

See also: – **RTFFromRange:documentAttributes:**, – **RTFDFileWrapperFromRange: documentAttributes:**

RTFFromRange:documentAttributes:

– (NSData *)**RTFFromRange:**(NSRange)*aRange* **documentAttributes:** (NSDictionary *)*docAttributes*

Returns an NSData object that contains an RTF stream corresponding to the characters and attributes within *aRange*, omitting all attachment attributes. Also writes the document-level attributes in *docAttributes*, as explained in the class cluster description under the "RTF Document Attributes" heading. If there are no document-level attributes, *docAttributes* can be **nil**. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

When writing data to the pasteboard, you can use the NSData object as the first argument to NSPasteboard's **setData:forType:** method, with a second argument of NSRTFPboardType.

Although this method strips attachments, it leaves the attachment characters in the text itself. NSText's **RTFFromRange:** method, on the other hand, does strip attachment characters when extracting RTF.

See also: – **RTFDFromRange:documentAttributes:**, – **RTFDFileWrapperFromRange: documentAttributes:**

rulerAttributesInRange:

– (NSDictionary *)**rulerAttributesInRange:**(NSRange)*aRange*

Returns the ruler (paragraph) attributes in effect for the characters within *aRange*. The only ruler attribute currently defined is that named by NSParagraphStyleAttributeName. Use this method to obtain attributes that are to be copied or pasted with “copy ruler” operations. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver’s characters.

See also: – **fontAttributesInRange:**

size

– (NSSize)**size**

Returns the bounding box of the marks that the receiver draws.

See also: – **drawAtPoint:**, – **drawInRect:**

NSMutableAttributedString

Inherits From: NSAttributedString : NSObject

Declared In: AppKit/NSAttributedString.h
AppKit/NSStringDrawing.h
AppKit/NSTextAttachment.h

Class Description

Additions to the NSMutableAttributedString class primarily involve setting graphical attributes, such as font, super- or subscripting, and alignment, and making these attributes consistent after changes. See the class cluster description for more information.

Method Types

Changing attributes	– applyFontTraits:range: – setAlignment:range: – subscriptRange: – superscriptRange: – unscriptRange:
Updating attachment contents	– updateAttachmentsFromPath:
Fixing attributes after changes	– fixAttributesInRange: – fixAttachmentAttributeInRange: – fixFontAttributeInRange: – fixParagraphStyleAttributeInRange:

Instance Methods

applyFontTraits:range:

– (void)**applyFontTraits:**(NSFontTraitMask)*mask* **range:**(NSRange)*aRange*

Apply the font attributes specified by *mask* to the characters in *aRange*. See the NSFontManager class specification for a description of the font traits available. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

See also: – **setAlignment:range:**

fixAttachmentAttributeInRange:

– (void)**fixAttachmentAttributeInRange:**(NSRange)*aRange*

Cleans up attachment attributes in *aRange*, removing all attachment attributes assigned to characters other than NSAttachmentCharacter. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

See also: – **fixFontAttributeInRange:**, – **fixParagraphStyleAttributeInRange:**, – **fixAttributesInRange:**

fixAttributesInRange:

– (void)**fixAttributesInRange:**(NSRange)*aRange*

Invokes the other **fix...** methods, allowing you to clean up an attributed string with a single message. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

See also: – **fixAttachmentAttributeInRange:**, – **fixFontAttributeInRange:**, – **fixParagraphStyleAttributeInRange:**

fixFontAttributeInRange:

– (void)**fixFontAttributeInRange:**(NSRange)*aRange*

Fixes the font attribute in *aRange*, assigning default fonts to characters with illegal fonts for their scripts and otherwise correcting font attribute assignments. For example, Kanji characters in assigned a Latin font are reassigned an appropriate Kanji font. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

See also: – **fixParagraphStyleAttributeInRange:**, – **fixAttachmentAttributeInRange:**, – **fixAttributesInRange:**

fixParagraphStyleAttributeInRange:

– (void)**fixParagraphStyleAttributeInRange:**(NSRange)*aRange*

Fixes the paragraph style attributes in *aRange*, assigning the first paragraph style attribute value in each paragraph to all characters of the paragraph. This method extends the range as needed to cover the last paragraph partially contained. A paragraph is delimited by any of these characters, the longest possible sequence being preferred to any shorter:

U+000D (**\r** or CR)	U+2028 (Unicode line separator)
U+000A (**\n** or LF)	U+2029 (Unicode paragraph separator) **\r\n**, in that order (also known as CRLF)

Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

See also: – **fixFontAttributeInRange:**, – **fixAttachmentAttributeInRange:**, – **fixAttributesInRange:**

setAlignment:range:

– (void)**setAlignment:**(NSTextAlignment)*alignment* **range:**(NSRange)*aRange*

Sets the alignment characteristic of the paragraph style attribute for the characters in *aRange* to *alignment*. When attribute fixing takes place, this change will only affect paragraphs whose first character was included in *aRange*. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

See also: – **applyFontTraits:range:**, – **fixParagraphStyleAttributeInRange:**

subscriptRange:

– (void)**subscriptRange:**(NSRange)*aRange*

Decrements the value of the superscript attribute for characters in *aRange* by 1. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

See also: – **superscriptRange:**, – **unscriptRange:**

superscriptRange:

– (void)**superscriptRange:**(NSRange)*aRange*

Increments the value of the superscript attribute for characters in *aRange* by 1. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

See also: – **subscriptRange:**, – **unscriptRange:**

unscriptRange:

– (void)**unscriptRange:**(NSRange)*aRange*

Removes the superscript attribute from the characters in *aRange*. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

See also: – **subscriptRange:**, – **superscriptRange:**

updateAttachmentsFromPath:

– (void)**updateAttachmentsFromPath:**(NSString *)*path*

Updates all attachments based on files contained in the RTFD file package at *path*.

See also: – **updateFromPath:** (NSFileWrapper)

NSBezierPath

Inherits From:	NSObject
Conforms To:	NSCoding NSCopying NSObject (NSObject)
Declared In:	AppKit/NSBezierPath.h

Class Description

An NSBezierPath object allows you to create paths using PostScript-style commands. Paths consist of straight and curved line segments joined together. Paths can form recognizable shapes such as rectangles, ovals, arcs, and glyphs; they can also form complex polygons using either straight or curved line segments. A single path can be closed by connecting its two endpoints or it can be left open.

An NSBezierPath object can contain multiple disconnected paths, whether they are closed or open. Each of these paths is referred to as a *subpath* of the NSBezierPath object. The subpaths of an NSBezierPath object must be manipulated as a group. The only way to manipulate subpaths individually is to create separate NSBezierPath objects for each.

For a given NSBezierPath object, you can stroke the path's outline or fill the region occupied by the path. You can also use the path as a clipping region for views or other regions. Using methods of NSBezierPath, you can also perform hit detection on the filled or stroked path. Hit detection is needed to implement interactive graphics, as in rubberbanding and dragging operations.

Constructing Paths

You can create an instance of NSBezierPath using either of the class methods **bezierPath** or **bezierPathWithRect:**. The **bezierPath** method initializes a new bezier-path object with an empty path while the **bezierPathWithRect:** method creates a path with the specified rectangle. (You can also allocate memory for a new instance of NSBezierPath and use the default initializer, the **init** method, to initialize it to an empty path).

To add path information to an NSBezierPath object, you must invoke a sequence of *path construction methods* such as the **moveToPoint:**, **lineToPoint:**, and **curveToPoint:controlPoint1:controlPoint2:** methods. For example, to create a polygon path, send a **moveToPoint:** message, followed by a series of **lineToPoint:** messages, to the NSBezierPath object. When you are done adding points to the path and want to form a closed path, send a **closePath** message to connect the last point back to the starting point.

The order in which path construction methods are invoked is significant. Line segments *connect* only if they are defined consecutively and the second segment starts where the first segment ends. A path may be made

up of one or more disconnected subpaths, which can themselves be either open or closed. The *current point* is the last point of the most recently added line segment.

Most construction methods implicitly use the current point as the starting point of the next segment. If you want to create a new subpath, you must explicitly invoke **moveToPoint:** first. For example, **lineToPoint:** adds a line segment from the current point to the specified point. Some methods may implicitly invoke **moveToPoint:**, thereby creating a new subpath automatically. See the method descriptions for more information.

Convenience methods are provided for appending paths and common shapes to an existing path. The new path is usually discontiguous from the receiver's original path, although the **appendBezierPathWithPoints:count:** method creates a contiguous path from the specified points. Use the **appendBezierPath...** methods to append a path to an NSBezierPath object as in this example which uses **appendBezierPathWithOvalInRect:** to create a circle:

```
NSRect aRect =
NSMakeRect(0.0, 0.0, 50.0, 50.0);
aPath = [[NSBezierPath bezierPath] appendBezierPathWithOvalInRect:aRect];
```

Path Elements

No matter which construction methods you use, all paths are reduced to a sequence of data points and common element types corresponding to the methods: **moveToPoint:**, **lineToPoint:**, **curveToPoint: controlPoint1:controlPoint2:** and **closePath**. The element types are as follows can be specified with the following constants from the enumerated type NSBezierPathElementType:

- NSBezierPathElementMoveTo
- NSBezierPathElementLineTo
- NSBezierPathElementCurveTo
- NSBezierPathElementClose

Every element except NSBezierPathElementClose has at least one data point associated with it. The only element that has more than one data point is NSBezierPathElementCurveTo (which maintains additional control points to define the shape of the curve). NSBezierPath defines several methods for obtaining information about the path elements (and their associated points) directly, including **pointAtIndex:**, **elementAtIndex:**, and **elementTypeAtIndex:associatedPoints:** among others. You could use these methods together with the **removeLastElement** method to break down a path and reconstruct it point by point.

Drawing Paths

You typically render NSBezierPath objects inside an NSView's **drawRect:** method (unless you are drawing outside of the bounds of an NSView object, as in a rubberbanding operation). At the time the **drawRect:** method is invoked, the focus is locked on the view and all drawing operations are clipped to that view. Therefore, most of the time you will want to construct paths whose points are specified in the view's coordinate system. You could also construct a path using an arbitrary coordinate system and transform the

path to a view's coordinate system using an NSAffineTransform object. An NSAffineTransform object can translate, scale, and rotate paths (see the NSAffineTransform class specification for details).

Before filling or stroking an NSBezierPath object, you should set the graphics attributes to use for the path. You can use the **set...** methods of NSBezierPath to set such attributes as the line cap style, line join style, line width, miter limit, curve flatness, and halftone phase. Other attributes must be set using the appropriate objects. For example, you set the color in the current graphics context by sending **set** to an NSColor object.

You can use either the **stroke**, or **fill** methods to render a path. The **stroke** method draws a line along the receiver's path using the color, line width, cap and join styles, and curve flatness drawing attributes in the current graphics context. The **fill** method renders the path by painting the region enclosed by the path and uses the color and curve flatness attributes. The **fill** method will perform a close operation (invoke **closePath**) if the path is not already closed. (A subpath is *closed* if the ending point is connected to the starting point, otherwise the subpath is *opened*).

As a convenience, some class methods are provided for drawing immediate shapes without the creation of a NSBezierPath object. For example, use the **fillRect:** and **strokeRect:** class methods to draw a filled rectangle or outline of a rectangle, and use the **strokeLineFromPoint:toPoint:** class method to draw a line segment.

Winding Rules and Filling Paths

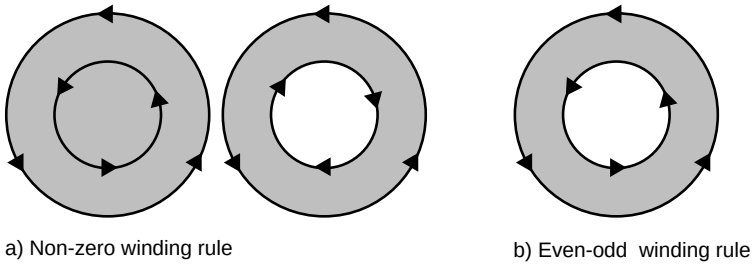
Simple paths like an oval or rectangle are easy to fill; however, there are several ways to fill complex paths that contain intersecting line segments or that contain a subpath enclosed by another subpath (i.e., a doughnut shape). You can specify how complex paths are filled using the **setWindingRule:** method and the constants NSWindingRuleNonZero or NSWindingRuleEvenOdd.The rules that govern the effects of each constant are as follows:

Winding Rule	Description
NSWindingRuleNonZero	A point is outside if drawing a ray from that point in any direction results in a crossing count of 0, where crossing a left-to-right path adds 1 and crossing a right-to-left path subtracts 1. Otherwise, the point is inside.
NSWindingRuleEvenOdd	A point is inside if drawing a ray from that point in any direction and counting the number of path segments that the ray crosses is odd, otherwise the point is outside. Inside points are filled, outside points are not.

For example, given a path with two nested circles, Figure 1 shows the results of using each winding rule. If the winding rule is NSWindingRuleNonZero, the direction of the paths is used to determine whether or not a segment should be filled. When the two paths are traveling in the same direction, the entire area of both circles is filled, but when the paths travel counter to each other the interior circle is left unfilled. When the

winding rule is NSWindingRuleEvenOdd, the interior circle is left unfilled regardless of which direction the paths travel.

Figure 1 Winding rule examples



Hit Detection

Hit detection is necessary if you want the user to be able to select your paths or graphical shapes. The Application Kit notifies your application know if the user clicked within the bounds of an NSView object by invoking one of the NSResponder **mouse...** methods. You use the NSBezierPath **isHitByPoint:** method to determine if the user clicked a filled path, and the **isStrokeHitByPoint:** method to determine if the user clicked on a stroked path. The **isHitByRect:** and **isStrokeHitByRect:** methods are useful if the user has selected a region (for example, by rubberbanding a rectangle) and you want to determine which paths lie inside that region.

Setting Path Styles

NSBezierPath includes several methods for setting the current drawing style to be used for rendering paths. Most drawing styles are global attributes, set using class methods of NSBezierPath. The one exception to this is the winding rule attribute, which is local to a particular path. The global attributes include path-related attributes such as the line width, miter limit, line join style, line cap style, and curve flatness. For information about setting the winding rule of a path, see “Winding Rules and Filling Paths”.

To set other attributes of the path, such as the color, you must use the methods of the appropriate class. For example, to set the color of the path, you would need to create a new instance of NSColor and invoke its **set** method.

Adopted Protocols

NSCoding

– encodeWithCoder:
– initWithCoder:

NSCopying

– copyWithZone:

Method Types

Creating an NSBezierPath object

+ bezierPath
+ bezierPathWithRect:

Contructing paths

– moveToPoint:
– lineToPoint:
– curveToPoint:controlPoint1:controlPoint2:
– closePath
– reset
– relativeMoveToPoint:
– relativeLineToPoint:
– relativeCurveToPoint:controlPoint1:controlPoint2:

Appending paths and some common shapes

– appendBezierPath:
– appendBezierPathWithRect:
– appendBezierPathWithPoints:count:
– appendBezierPathWithOvalInRect:
– appendBezierPathWithArcWithCenter:radius:startAngle: endAngle:
– appendBezierPathWithGlyph:inFont:
– appendBezierPathWithGlyphs:count:inFont:
– appendBezierPathWithPackedGlyphs:

Setting attributes

– setWindingRule:
– windingRule
+ setLineCapStyle:
+ setLineJoinStyle:
+ setLineWidth:
+ setMiterLimit:
+ setFlatness:
+ setHalftonePhase:

Drawing paths

– stroke
– fill
+ fillRect:
+ strokeRect:
+ strokeLineFromPoint:toPoint:
+ drawPackedGlyphs:atPoint:

Clipping paths

– addClip
– setClip
+ clipRect:

Hit detection

– isHitByPoint:
– isHitByRect:
– isHitByPath:
– isStrokeHitByPoint:
– isStrokeHitByRect:
– isStrokeHitByPath:

Querying paths

– bounds
– controlPointBounds
– currentPoint

Accessing elements of a path

– elementCount
– elementTypeAtIndex:
– elementTypeAtIndex:associatedPoints:
– removeLastElement
– pointCount
– pointAtIndex:
– setPointAtIndex:toPoint:
– pointIndexForPathElementIndex:
– pathElementIndexForPointIndex:

Caching paths

– cachesBezierPath
– setCachesBezierPath:

Class Methods

bezierPath

+ (NSBezierPath *)**bezierPath**

Creates and returns a new NSBezierPath object. The path is initially empty.

bezierPathWithRect:

+ (NSBezierPath *)**bezierPathWithRect:**(NSRect)*aRect*

Creates and returns a new NSBezierPath object with a rectangular path specified by *aRect*. The path starts at the origin of *aRect*, and is constructed counterclockwise.

See also: **+ bezierPath**, – **appendBezierPathWithRect:**, + **fillRect:**, + **strokeRect:**

clipRect:

+ (void)**clipRect:**(NSRect)*aRect*

Intersects the current clipping path, stored in the current graphics context, with the rectangle referred to by *aRect*, and replaces the current clipping path with the resulting path.

See also: – **addClip**, – **setClip**

drawPackedGlyphs:atPoint:

+ (void)**drawPackedGlyphs:**(const char *)*packedGlyphs* **atPoint:**(NSPoint)*aPoint*

Draws the characters in *packedGlyphs* at *aPoint* in the current graphic context's coordinate system. The glyphs are drawn immediately.

See also: – **appendBezierPathWithGlyph:inFont:**, – **appendBezierPathWithGlyphs:count:inFont:**, – **appendBezierPathWithPackedGlyphs:**, – **set** (NSColor)

fillRect:

+ (void)**fillRect:**(NSRect)*aRect*

Fills a rectangular path specified by *aRect* with the current color (stored in the current graphics context).

See also: – **appendBezierPathWithRect:**, + **bezierPathWithRect:**, – **set** (NSColor), + **strokeRect:**

setFlatness:

+ (void)**setFlatness:**(float)*flatness*

Sets the current graphics context's flatness attribute to *flatness*. The *flatness attribute* is the accuracy (or smoothness) with which curves are rendered. *flatness* is the maximum error tolerance, measured in pixels, where smaller numbers give smoother curves at the expense of more computation. The exact interpretation may vary slightly on different rendering devices.

The default flatness value for a graphic context is 1.0. Flatness values that are less than 0.2 or greater than 100 are normalized to those boundaries.

setHalftonePhase:

+ (void)**setHalftonePhase:**(NSPoint)*aPoint*

Sets the current graphics context's halftone phase to *aPoint*. The *halftone phase* is a shift in the device-space alignment of halftone and pattern cells to compensate for window system operations that involve scrolling. For example, if your application performs a scroll by (*dx*, *dy*) pixels in device space, you should simply add *dx* and *dy* to the halftone phase parameters.

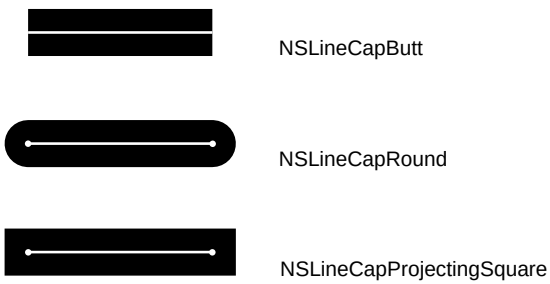
This is a device dependent property. The default value is (0, 0).

setLineCapStyle:

+ (void)**setLineCapStyle:**(NSLineCapStyle)*lineCap*

Sets the current graphics context's line cap style to *lineCap*. The *line cap style* specifies the shape of the endpoints of an open path when stroked. Figure 2 shows the appearance of the available line cap styles.

Figure 2 Line cap styles



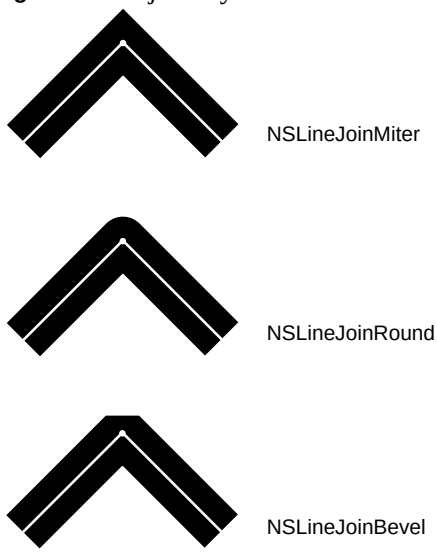
See also: + **setLineJoinStyle:**, + **setLineWidth:**

setLineJoinStyle:

+ (void)**setLineJoinStyle:**(NSLineJoinStyle)*lineJoinStyle*

Sets the current graphics context's line join style to *lineJoinStyle*. The *line join style* specifies the shape of the joints between connected segments of a stroked path. Figure 3 shows the appearance of the available line join styles.

Figure 3 Line join styles



See also: + **setLineCapStyle:**, + **setLineWidth:**, + **setMiterLimit:**

setLineWidth:

+ (void)**setLineWidth:**(float)*width*

Sets the current graphics context's line width to *width* points. The *line width* is the thickness of stroked paths. A width of zero is interpreted as the thinnest line that can be rendered on a particular device. The actual rendered line width may vary from *width* by as much as two device pixels, depending on the position of the line with respect to the pixel grid. The width of the line may also be affected by scaling factors specified in the current transformation matrix of the active graphics context.

See also: + **setLineCapStyle:**, + **setLineJoinStyle:**

setMiterLimit:

+ (void)**setMiterLimit:**(float)*limit*

Sets the current graphics context's miter limit to *limit*. Setting the miter limit avoids spikes produced by line segments that join at sharp angles. If the ratio of the miter length—the diagonal length of the miter—to the line thickness exceeds the miter limit, the corner is treated as a bevel join instead of a miter join. The default miter limit value is 10, which cuts off miters at angles less than 11 degrees.

See also: + **setLineJoinStyle:**

strokeLineFromPoint:toPoint:

+ (void)**strokeLineFromPoint:**(NSPoint)*point1* **toPoint:**(NSPoint)*point2*

Strokes a line from *point1* to *point2* using the current graphics context's drawing attributes (for example, color, line cap style, and line width).

See also: – **lineToPoint:**, – **moveToPoint:**, **+ setLineCapStyle:**, **+ setLineWidth:**, **– stroke**

strokeRect:

+ (void)**strokeRect:**(NSRect)*aRect*

Strokes a rectangular path specified by *aRect* using the current graphics context's drawing style and color. The path is stroked beginning at the rectangle's origin and proceeding in a counterclockwise direction.

See also: – **appendBezierPathWithRect:**,+ **bezierPathWithRect:**, + **fillRect:,** – **set** (NSColor), **+ setLineJoinStyle:**, **+ setLineWidth:**

Instance Methods

addClip

– (void)**addClip**

Intersects the current clipping path, stored in the current graphics context, with the receiver's path, and replaces the current clipping path with the resulting path. The current winding rule is applied to determine the clipping area of the receiver. This method does not affect the receiver's path.

See also: + clipRect:,– setClip

appendBezierPath:

– (void)**appendBezierPath:**(NSBezierPath *)*aPath*

Appends *aPath* to the receiver's path. This method adds the operations used to create *aPath* to the end of the receiver's path. This method does not explicitly try to connect the two paths, although the operations in *aPath* may still cause this effect.

appendBezierPathWithArcWithCenter:radius:startAngle:endAngle:

– (void)**appendBezierPathWithArcWithCenter:**(NSPoint)*center*
radius:(float)*radius*
startAngle:(float)*startAngle*
endAngle:(float)*endAngle*

Appends an arc of a circle to the receiver's path. The circle is centered at *center* with radius *radius*. The arc lies on the perimeter of the circle, between *startAngle* and *endAngle*, measured in degrees counterclockwise from the x-axis.

appendBezierPathWithGlyph:inFont:

– (void)**appendBezierPathWithGlyph:**(NSGlyph)*aGlyph* **inFont:**(NSFont *)*fontObj*

Appends an outline of *aGlyph* in *fontObj* to the receiver's path. If *aGlyph* is not encoded in *fontObj*—that is, the font does not have an entry for the specified glyph—then no path is appended to the receiver.

See also: **– appendBezierPathWithGlyphs:count:inFont:**, **– appendBezierPathWithPackedGlyphs:**, **+ drawPackedGlyphs:atPoint:**

appendBezierPathWithGlyphs:count:inFont:

– (void)**appendBezierPathWithGlyphs:**(NSGlyph *)*glyphs*
count:(int)*count*
inFont:(NSFont *)*fontObj*

Appends the outlines of the NSGlyphs in the *glyphs* array to the receiver's path. The *glyphs* must be encoded using the font in *fontObj*. The *count* parameter specifies the number of NSGlyphs in *glyphs*. If an NSGlyph is not encoded in *fontObj*—that is, the font does not have an entry for the specified glyph—then no path is appended for that glyph.

See also: **– appendBezierPathWithGlyph:inFont:**, **– appendBezierPathWithPackedGlyphs:**, **+ drawPackedGlyphs:atPoint:**

appendBezierPathWithOvalInRect:

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

Appends an oval path, inscribed in the rectangle *aRect*, to the receiver's path. If *aRect* specifies a square, the inscribed path is a circle. The inscribed path starts at the top center of *aRect* and arc segments are constructed counterclockwise to complete the oval.

appendBezierPathWithPackedGlyphs:

– **appendBezierPathWithPackedGlyphs:**(const char *)*packedGlyphs*

Appends the *packedGlyphs* to the receiver's path. You should not use this method directly. Instead, use the **appendBezierPathWithGlyph:inFont:** and **appendBezierPathWithGlyphs:count:inFont:** methods to append glyphs to a path.

See also: – **appendBezierPathWithGlyph:inFont:**, – **appendBezierPathWithGlyphs:count:inFont:**, + **drawPackedGlyphs:atPoint:**

appendBezierPathWithPoints:count:

– (void)**appendBezierPathWithPoints:**(NSPoint *)*points* **count:**(int)*count*

Appends a series of line segments with *count* number of vertices in *points* to the receiver's path. If the receiver's path is empty, this method simply creates a new path from the supplied points. If the receiver contains an existing path (or even a single point), this method appends the points to the existing path, creating a line segment from the last point in the receiver's path to the first point in the *points* array.

This method does not close the receiver's path. If you wish to create a closed path, you must do so by explicitly invoking the receiver's **closePath** method.

See also: **– closePath**

appendBezierPathWithRect:

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

Appends a rectangular path, specified by *aRect*, to the receiver's path. The path starts at the origin of *aRect* and line segments are added proceeding counterclockwise from the origin, ending with a message to **closePath** to complete the path.

See also: + **bezierPathWithRect:**, + **fillRect:**, + **strokeRect:**, **– closePath**

bounds

– (NSRect)**bounds**

Returns the bounding box of the receiver's path. If the path contains curve segments, the bounding box encloses the curve but may not enclose the control points used to calculate the curve.

See also: – **controlPointBounds**

cachesBezierPath

– (BOOL)**cachesBezierPath**

Returns YES if this object maintains a cached image of its path, otherwise returns NO. The cached image is stored in a display postscript user object.

See also: **– setCachesBezierPath:**

closePath

– (void)**closePath**

Closes the most recently added subpath by appending a straight line segment from the current point to the subpath's starting point. A *subpath* is a sequence of connected segments; a path may be made up of one or more disconnected subpaths. The *current point* is the ending point in the most recently added segment.

See also: – **fill**

controlPointBounds

– (NSRect)**controlPointBounds**

Returns the bounding box of the receiver's path including any control points. If the path contains curve segments, the bounding box encloses the control points of the curves as well as the curves themselves.

See also: – **bounds**

currentPoint

– (NSPoint)**currentPoint**

Returns the path's *current point* (the trailing point or ending point in the most recently added segment). Raises NSGenericException if the path is empty.

See also: **– closePath,** – **curveToPoint:controlPoint1:controlPoint2:,** – **lineToPoint:**, – **moveToPoint:**, **– reset**

curveToPoint:controlPoint1:controlPoint2:

– (void)**curveToPoint:**(NSPoint)*aPoint*
controlPoint1:(NSPoint)*controlPoint1*
controlPoint2:(NSPoint)*controlPoint2*

Adds a Bezier cubic curve to the receiver's path from the current point to *aPoint*, using *controlPoint1* and *controlPoint2* as the Bezier cubic control points. The *current point* is the ending point in the most recently added segment. To create a relative curve, use the **relativeCurveToPoint:controlPoint1:controlPoint2:** method.

See also: – **closePath**, – **lineToPoint:**, – **moveToPoint:**, **+ setFlatness:**, **– relativeCurveToPoint: controlPoint1:controlPoint2:**

elementCount

– (int)**elementCount**

Returns the number of element types currently stored by the receiver's path. Each element type corresponds to one of the operations specified by the NSBezierPathElementType enumerated type.

See also: – **elementTypeAtIndex:**, – **elementTypeAtIndex:associatedPoints:**, – **removeLastElement**

elementTypeAtIndex:

– (NSBezierPathElementType)**elementTypeAtIndex:**(int)*index*

Returns the element type at *index*. Element types describe the operations that make up a path and include such basic commands as moving to a specific point, creating a line segment, creating a curve, or closing the path. The element types are stored in the order of execution and so the *index* parameter specifies the operation at a given moment in the life of the path.

See also: – **elementCount**, – **elementTypeAtIndex:associatedPoints:**, – **removeLastElement**

elementTypeAtIndex:associatedPoints:

– (NSBezierPathElementType)**elementTypeAtIndex:**(int)*index* **associatedPoints:**(NSPoint *)*points*

Returns the element type at *index* and returns any points associated with that element type in the *points* parameter. You must allocate an array of NSPoint objects large enough to hold the number of points for the given element and pass it into the *points* parameter. Move and line segment operations return one point. Curve operations return three points. Close path operations return zero points.

See also: – **elementCount**, – **elementTypeAtIndex:**, – **removeLastElement**

fill

– (void)**fill**

Renders the receiver's path by painting the region enclosed by the path. Uses the winding rule, specified by inoking **setWindingRule:** and the current graphics context's color to fill the path. Closes any open subpaths. A *subpath* is a sequence of connected segments; a path may be made up of one or more disconnected subpaths. A subpath is *closed* if the ending point is connected to the starting point (as in a polygon).

See also: – **set** (NSColor), – **setWindingRule:**, – **stroke**, – **windingRule**

isHitByPath:

– (BOOL)**isHitByPath:**(NSBezierPath *)*aBezierPath*

Returns YES if any part of *aBezierPath* intersects the receiver's path, otherwise returns NO.

See also: – **isHitByPoint:**, – **isHitByRect:**, – **isStrokeHitByPath:**, **– bounds**

isHitByPoint:

– (BOOL)**isHitByPoint:**(NSPoint)*aPoint*

Returns YES if *aPoint* lies within the filled area of the receiver's path, otherwise returns NO.

See also: – **isHitByPath:**, – **isHitByRect:**, – **isStrokeHitByPoint:**, **– bounds**

isHitByRect:

– (BOOL)**isHitByRect:**(NSRect)*aRect*

Returns YES if *aRect* intersects the receiver's path, otherwise returns NO.

See also: – **isHitByPath:**, – **isHitByPoint:**, – **isStrokeHitByRect:**, **– bounds**

isStrokeHitByPath:

– (BOOL)**isStrokeHitByPath:**(NSBezierpath *)*aBezierPath*

Returns YES if the path of *aBezierPath* intersects any point on the receiver's path, otherwise returns NO.

See also: – **isStrokeHitByPoint:**, – **isStrokeHitByRect:**, – **isHitByPath:**

isStrokeHitByPoint:

– (BOOL)**isStrokeHitByPoint:**(NSPoint)*aPoint*

Returns YES if *aPoint* lies on the receiver's path, otherwise returns NO.

See also: – **isStrokeHitByPath:**, – **isStrokeHitByRect:**, – **isHitByPoint:**

isStrokeHitByRect:

– (BOOL)**isStrokeHitByRect:**(NSRect)*aRect*

Returns YES if *aRect* intersects the receiver's path, otherwise returns NO.

See also: – **isStrokeHitByPath:**, – **isStrokeHitByPoint:**, – **isHitByRect:**

lineToPoint:

– (void)**lineToPoint:**(NSPoint)*aPoint*

Appends a straight line to the receiver's path from the current point to *aPoint*. The current point is the last point in the receiver's most recently added segment.

See also: **– closePath,** – **curveToPoint:controlPoint1:controlPoint2:,** – **moveToPoint:**

moveToPoint:

– (void)**moveToPoint:**(NSPoint)*aPoint*

Moves the receiver's current point to *aPoint*, starting a new subpath, without adding any line segments. A *subpath* is a sequence of connected segments; a path may be made up of one or more disconnected subpaths. The *current point* is the ending point in the most recently added segment.

See also: **– closePath,** – **curveToPoint:controlPoint1:controlPoint2:,** – **lineToPoint:**

pathElementIndexForPointIndex:

– (int)**pathElementIndexForPointIndex:**(int)*index*

Given the *index* of a point in the path, this method returns the index of the element type that corresponds to that point. The *index* parameter can specify either a point on the path or a control point used to define a curve. The returned value can be passed to the **elementTypeAtIndex:** method to determine the element type that operates on the point.

See also: – **pointAtIndex:**, – **pointCount**, – **pointIndexForPathElementIndex:**, – **setPointAtIndex: toPoint:**, **– elementTypeAtIndex:**

pointAtIndex:

– (NSPoint)**pointAtIndex:**(int)*index*

Returns the point at the specified *index*. The point may be either a point on the path or a control point used to define a curve.

See also: – **pathElementIndexForPointIndex:**, – **pointCount**, – **pointIndexForPathElementIndex:**, – **setPointAtIndex:toPoint:**

pointCount

– (int)**pointCount**

Returns the number of points used to define the receiver's path. The number of points includes points on the path and control points used to define curves.

See also: – **pathElementIndexForPointIndex:**, – **pointAtIndex:**, – **pointIndexForPathElementIndex:**, – **setPointAtIndex:toPoint:**

pointIndexForPathElementIndex:

– (int)**pointIndexForPathElementIndex:**(int)*index*

Returns the array location of the first point associated with the element type at *index*. If the element type uses more than one point, this method returns the index of only the first point.

See also: – **pathElementIndexForPointIndex:**, – **pointAtIndex:**, – **pointCount**, – **setPointAtIndex: toPoint:**

relativeCurveToPoint:controlPoint1:controlPoint2:

– (void)**relativeCurveToPoint:**(NSPoint)*aPoint*
controlPoint1:(NSPoint)*controlPoint1*
controlPoint2:(NSPoint)*controlPoint2*

Adds a Bezier cubic curve to the receiver's path from the current point to a new location, which is specified as a relative distance from the current point. (The control points are similarly specified as relative distances from the current point.) The *aPoint* parameter specifies the end point of the curve as a relative distance from the current point. The *controlPoint1* and *controlPoint2* parameters specify the location of the two control points as relative distances from the current point. Raises NSGenericException if the path is empty.

See also: – **closePath**, – **curveToPoint:controlPoint1:controlPoint2:**, – **relativeLineToPoint:**, – **relativeMoveToPoint:**

relativeLineToPoint:

– (void)**relativeLineToPoint:**(NSPoint)*aPoint*

Appends a straight line to the receiver's path from the current point to *aPoint*, which is specified as a relative distance from the current point. Raises NSGenericException if the path is empty.

See also: – **closePath**, – **lineToPoint:**, – **relativeCurveToPoint:controlPoint1:controlPoint2:**, – **relativeMoveToPoint:**

relativeMoveToPoint:

– (void)**relativeMoveToPoint:**(NSPoint)*aPoint*

Moves the receiver's current point to a new point, which is specified by the parameter *aPoint* as a relative distance from the current point. This method starts a new subpath without adding any line segments.

See also: – **closePath**, – **moveToPoint:**, – **relativeCurveToPoint:controlPoint1:controlPoint2:**, – **relativeLineToPoint:**

removeLastElement

– (void)**removeLastElement**

Undoes the most recent path operation. This method removes only one simple operation, such as moving the current point, creating a line segment, or creating a curve. If you last created a new rectangle or oval, this method removes only one line segment or curve from the added path.

See also: – **elementCount**, – **elementTypeAtIndex:**, – **elementTypeAtIndex:associatedPoints:**

reset

– (void)**reset**

Sets the receiver's path to an empty path, a path containing no subpaths. After invoking this method the current point is undefined.

setCachesBezierPath:

– (void)**setCachesBezierPath:**(BOOL)*flag*

Sets whether the receiver should cache its path information. Caching improves subsequent drawing times but requires extra memory to store the cached path representation. If caching is being turned on (*flag* is

YES), the receiver's cache is marked as needing to be calculated. Otherwise, if caching is being turned off, any existing cached data is deleted.

See also: – **cachesBezierPath**

setClip

– (void)**setClip**

Replace the current clipping path with the area inside this path as determined by the winding rule. This is not a preferred method of adjusting the clipping path, as it may expand the clipping path beyond the bounds set by the enclosing NSView. The graphics state should be saved and restored before and after invoking this method.

See also: – **addClip**, + **clipRect:**, – **saveGraphicsState (NSGraphicsContext)**

setPointAtIndex:toPoint:

– (void)**setPointAtIndex:**(int)*index* **toPoint:**(NSPoint)*aPoint*

Changes the value of the point at *index* to the new point specified by *aPoint*. The *index* parameter is zero-based and refers to the array of points used to specify the path. Invoking this method marks any cached data as needing to be recalculated.

See also: – **pathElementIndexForPointIndex:**, – **pointAtIndex:**, – **pointCount**, – **pointIndexForPathElementIndex:**

setWindingRule:

– (void)**setWindingRule:**(NSWindingRule)*aWindingRule*

Sets the winding rule used to fill the receiver's path, that is, paint the region enclosed by the path. Possible values for the *aWindingRule* parameter are NSWindingRuleNonZero or NSWindingRuleEvenOdd. See "Winding Rules and Filling Paths" for more information on how winding rules affect the appearance of filled paths.

See also: – **fill**, – **windingRule**

stroke

– (void)**stroke**

Draws a line along the receiver's path using the current graphic context's color and other drawing attributes (for example, line cap style, line join style, and line width). The drawn line is centered on the path with sides (specified by the **setLineWidth:** class method) parallel to the path segment.

See also: – **fill,** – **set** (NSColor), **+ setLineCapStyle:**, **+ setLineJoinStyle:**, **+ setLineWidth:**

windingRule

– (NSWindingRule)**windingRule**

Returns the winding rule used to fill the receiver's path, that is, paint the region enclosed by the path. Possible return values are NSWindingRuleNonZero or NSWindingRuleEvenOdd. See "Winding Rules and Filling Paths" for more information on how winding rules affect the appearance of filled paths.

See also: – **fill**, – **setWindingRule:**

NSBitmapImageRep

Inherits From: NSImageRep : NSObject

Conforms To: NSCoding (from NSImageRep)
NSCopying (from NSImageRep)
NSObject (from NSObject)

Declared In: AppKit/NSImage.h

Class Description

An NSBitmapImageRep is an object that can render an image from bitmap data. The data can be in Tag Image File Format (TIFF), Windows bitmap format (BMP), or it can be raw image data. If it's raw data, the object must be informed about the structure of the image—its size, the number of color components, the number of bits per sample, and so on—when it's first initialized. If it's TIFF or BMP data, the object can get this information from the various fields included with the data.

Although NSBitmapImageReps are often used indirectly, through instances of the NSImage class, they can also be used directly—for example to manipulate the bits of an image as you might need to do in a paint program.

Setting Up an NSBitmapImageRep

You pass bitmap data for an image to a new NSBitmapImageRep when you first initialize it. You can also create an NSBitmapImageRep from bitmap data that's read from a specified rectangle of a focused NSView.

Although the NSBitmapImageRep class inherits NSImageRep methods that set image attributes, these methods shouldn't be used. Instead, you should either allow the object to find out about the image from the fields included with the bitmap data, or use methods defined in this class to supply this information when the object is initialized.

TIFF Compression

TIFF data can be read and rendered after it has been compressed using any one of the four schemes briefly described below:

LZW	Compresses and decompresses without information loss, achieving compression ratios up to 5:1. It may be somewhat slower to compress and decompress than the PackBits scheme.

PackBits	Compresses and decompresses without information loss, but may not achieve the same compression ratios as LZW.
JPEG	Compresses and decompresses with some information loss, but can achieve compression ratios anywhere from 10:1 to 100:1. The ratio is determined by a user-settable factor ranging from 1.0 to 255.0, with higher factors yielding greater compression. More information is lost with greater compression, but 15:1 compression is safe for publication quality. Some images can be compressed even more. JPEG compression can be used only for images that specify at least 4 bits per sample.
CCITTFAX	Compresses and decompresses 1 bit gray-scale images using international fax compression standards CCITT3 and CCITT4.

An NSBitmapImageRep can also produce compressed TIFF data for its image using any of these schemes.

Method Types

Creating an NSBitmapImageRep

+ imageRepsWithData:
+ imageRepWithData:
– initWithBitmapDataPlanes:pixelsWide:pixelsHigh:bitsPerSample:samplesPerPixel:hasAlpha:isPlanar:colorSpaceName:bytesPerRow:bitsPerPixel:bytesPerRow:bitsPerPixel:
– initWithBitmapHandle:
– initWithData:
– initWithFocusedViewRect:
– initWithIconHandle:

Getting information about the image

– bitsPerPixel
– bytesPerPlane
– bytesPerRow
– isPlanar
– numberOfPlanes
– samplesPerPixel

Getting image data

– bitmapData
– getBitmapDataPlanes:

Producing a TIFF representation of the image

+ TIFFRepresentationOfImageRepsInArray:
+ TIFFRepresentationOfImageRepsInArray:usingCompression: factor:
– TIFFRepresentation
– TIFFRepresentationUsingCompression:factor:

Setting and checking compression types

+ getTIFFCompressionTypes:count:
+ localizedNameForTIFFCompressionType:
– canBeCompressedUsing:
– getCompression:factor:
– setCompression:factor:

Class Methods

getTIFFCompressionTypes:count:

+ (void)**getTIFFCompressionTypes:**(const NSTIFFCompression **)*list*
count:(int *)*numTypes*

Returns, by reference, an array of NSTIFFCompressions representing all available compression types that can be used when writing a TIFF image. The number of elements in list is represented by *numTypes*. *list* belongs to the NSBitmapImageRep class; it shouldn't be freed or altered.

The following compression types are supported:

Constant	Value	Usage
NSTIFFCompressionNone	1	
NSTIFFCompressionCCITTFAX3	3	1 bps images only
NSTIFFCompressionCCITTFAX4	4	1 bps images only
NSTIFFCompressionLZW	5	
NSTIFFCompressionJPEG	6	
NSTIFFCompressionNEXT	32766	Input only
NSTIFFCompressionPackBits	32773	
NSTIFFCompressionOldJPEG	32865	Input only

Note that not all compression types can be used for all images: NSTIFFCompressionNEXT can be used only to retrieve image data. Because future releases of OpenStep may include other compression types, always use this method to get the available compression types—for example, when you implement a user interface for selecting compression types.

See also: **+ localizedNameForTIFFCompressionType:**, **– canBeCompressedUsing:**

imageRepsWithData:

+ (NSArray *)**imageRepsWithData:**(NSData *)*bitmapData*

Creates and returns an array of initialized NSBitmapImageRep objects corresponding to the images in *bitmapData*. If NSBitmapImageRep is unable to interpret *bitmapData*, the returned array is empty. *bitmapData* can contain data in any supported bitmap format.

imageRepWithData:

+ (id)**imageRepWithData:**(NSData *)*bitmapData*

Creates and returns an initialized NSBitmapImageRep corresponding to the first image in *bitmapData*, or **nil** if NSBitmapImageRep is unable to interpret *bitmapData*. *bitmapData* can contain data in any supported bitmap format.

localizedNameForTIFFCompressionType:

+ (NSString *)**localizedNameForTIFFCompressionType:**(NSTIFFCompression)*compression*

Returns an autoreleased string containing the localized name for the compression type represented by *compression*, or **nil** if *compression* is unrecognized. Compression types are listed in the **getTIFFCompressionTypes:count:** class method description. When implementing a user interface for selecting TIFF compression types, use **getTIFFCompressionTypes:count:** to get the list of supported compression types, then use this method to get the localized names for each compression type.

See also: + **getTIFFCompressionTypes:count:**

TIFFRepresentationOfImageRepsInArray:

+ (NSData *)**TIFFRepresentationOfImageRepsInArray:**(NSArray *)*array*

Returns a TIFF representation of the images in *array*, using the compression that's returned by **getCompression:factor:** (if applicable).

If a problem is encountered during generation of the TIFF, **TIFFRepresentationOfImageRepsInArray** raises an NSTIFFException or an NSBadBitmapParametersException.

See also: **– TIFFRepresentation**

TIFFRepresentationOfImageRepsInArray:usingCompression:factor:

+ (NSData *)**TIFFRepresentationOfImageRepsInArray:**(NSArray *)*array*
usingCompression:(NSTIFFCompression)*compression*
factor:(float)*factor*

Returns a TIFF representation of the images in array, which are compressed using the specified compression type and factor. Legal values for *compression* can be found in **NSBitmapImageRep.h**, and are described in "Tiff Compression" in NSBitmapImageRep's class description. *factor* provides a hint for those compression types that implement variable compression ratios; currently only JPEG compression uses a compression factor. If your compression type doesn't implement variable compression ratios, or if it does and you don't want the image to be compressed, specify a compression factor of 0.0.

If the specified compression isn't applicable, no compression is used. If a problem is encountered during generation of the TIFF, **TIFFRepresentationOfImageRepsInArray:usingCompression:factor:** raises an NSTIFFException or an NSBadBitmapParametersException.

See also: **– TIFFRepresentationUsingCompression:factor:**

Instance Methods

bitmapData

– (unsigned char *)**bitmapData**

Returns a pointer to the bitmap data. If the data is planar, returns a pointer to the first plane.

See also: **– getBitmapDataPlanes:**

bitsPerPixel

– (int)**bitsPerPixel**

Returns the number of bits allocated for each pixel in each plane of data. This is normally equal to the number of bits per sample or, if the data is in meshed configuration, the number of bits per sample times the number of samples per pixel. It can be explicitly set to another value (in the **initWithBitmapDataPlanes: pixelsWide:pixelsHigh:...** method) in case extra memory is allocated for each pixel. This may be the case, for example, if pixel data is aligned on byte boundaries.

bytesPerPlane

– (int)**bytesPerPlane**

Returns the number of bytes in each plane or channel of data. This is calculated from the number of bytes per row and the height of the image.

See also: **– bytesPerRow**

bytesPerRow

– (int)**bytesPerRow**

Returns the minimum number of bytes required to specify a scan line (a single row of pixels spanning the width of the image) in each data plane. If not explicitly set to another value (in the **initWithBitmapDataPlanes:pixelsWide:pixelsHigh:...** method), this will be figured from the width of the image, the number of bits per sample, and, if the data is in a meshed configuration, the number of samples per pixel. It can be set to another value to indicate that each row of data is aligned on word or other boundaries.

See also: **– bytesPerPlane**

canBeCompressedUsing:

– (BOOL)**canBeCompressedUsing:**(NSTIFFCompression)*compression*

Tests whether the receiver can be compressed by compression type. Legal values for *compression* can be found in **NSBitmapImageRep.h**, and are described in “Tiff Compression” in the class description. This method returns YES if the receiver’s data matches compression; for example, if compression is NSTIFFCompressionCCITTFAX3, then the data must be one bit-per-sample and one sample-per-pixel. It returns NO if the data doesn’t match compression or if compression is unsupported.

See also: + **getTIFFCompressionTypes:count:**

getBitmapDataPlanes:

– (void)**getBitmapDataPlanes:**(unsigned char **)*data*

Provides access to bitmap data for the image separated into planes. *data* should be an array of five character pointers. If the bitmap data is in planar configuration, each pointer will be initialized to point to one of the data planes. If there are less than five planes, the remaining pointers will be set to NULL. If the bitmap data is in meshed configuration, only the first pointer will be initialized; the others will be NULL.

Color components in planar configuration are arranged in the expected order—for example, red before green before blue for RGB color. All color planes precede the coverage plane.

See also: **– isPlanar**

getCompression:factor:

– (void)**getCompression:**(NSTIFFCompression *)*compression* **factor:**(float *)*factor*

Returns by reference the receiver's compression type and compression factor. Use this method to get information on the compression type for the source image data. *compression* represents the compression type used on the data, and corresponds to one of the values returned by the class method getTIFFCompressionTypes:count:. factor is a value that is specific to the compression type; many types of compression don't support varying degrees of compression, and thus ignore factor. JPEG compression allows a compression factor ranging from 0.0 to 255.0, with 0.0 representing minimal compression.

initWithBitmapDataPlanes:pixelsWide:pixelsHigh:bitsPerSample:samplesPerPixel:hasAlpha:isPlanar:colorSpaceName:bytesPerRow:bitsPerPixel:

– (id)**initWithBitmapDataPlanes:**(unsigned char **)*planes*
pixelsWide:(int)*width*
pixelsHigh:(int)*height*
bitsPerSample:(int)*bps*
samplesPerPixel:(int)*spp*
hasAlpha:(BOOL)*alpha*
isPlanar:(BOOL)*isPlanar*
colorSpaceName:(NSString *)*colorSpaceName*
bytesPerRow:(int)*rowBytes*
bitsPerPixel:(int)*pixelBits*

Initializes the receiver, a newly allocated NSBitmapImageRep object, so that it can render the image specified in *planes* and described by the other arguments. If the object can't be initialized, this method frees it and returns **nil**. Otherwise, it returns the object (**self**).

planes is an array of character pointers, each of which points to a buffer containing raw image data. If the data is in planar configuration, each buffer holds one component—one plane—of the data. Color planes are arranged in the standard order—for example, red before green before blue for RGB color. All color planes precede the coverage plane.

If the data is in meshed configuration (*isPlanar* is NO), only the first buffer is read.

If *planes* is NULL or if it's an array of NULL pointers, this method allocates enough memory to hold the image described by the other arguments. You can then obtain pointers to this memory (with the **getBitmapDataPlanes:** or **bitmapData** method) and fill in the image data. In this case, the allocated memory will belong to the object and will be freed when it's freed.

If *planes* is not NULL and the array contains at least one data pointer, the object will only reference the image data; it won't copy it. The buffers won't be freed when the object is freed.

Each of the other arguments (besides *planes*) informs the NSBitmapImageRep object about the image. They're explained below:

- *width* and *height* specify the size of the image in pixels. The size in each direction must be greater than 0.
- *bps* (bits per sample) is the number of bits used to specify one pixel in a single component of the data. All components are assumed to have the same bits per sample.
- *spp* (samples per pixel) is the number of data components. It includes both color components and the coverage component (alpha), if present. Meaningful values range from 1 through 5. An image with cyan, magenta, yellow, and black (CMYK) color components plus a coverage component would have an *spp* of 5; a gray-scale image that lacks a coverage component would have an *spp* of 1.
- *alpha* should be YES if one of the components counted in the number of samples per pixel (*spp*) is a coverage component, and NO if there is no coverage component.
- *isPlanar* should be YES if the data components are laid out in a series of separate "planes" or channels ("planar configuration"), and NO if component values are interwoven in a single channel ("meshed configuration").

  For example, in meshed configuration, the red, green, blue, and coverage values for the first pixel of an image would precede the red, green, blue, and coverage values for the second pixel, and so on. In planar configuration, red values for all the pixels in the image would precede all green values, which would precede all blue values, which would precede all coverage values.
- *colorSpaceName* indicates how data values are to be interpreted. It should be one of the following enumerated values:

  NSCalibratedWhiteColorSpace

  NSCalibratedBlackColorSpace

  NSCalibratedRGBColorSpace

  NSDeviceWhiteColorSpace

  NSDeviceBlackColorSpace

  NSDeviceRGBColorSpace

  NSDeviceCMYKColorSpace

  NSNamedColorSpace

  NSCustomColorSpace
- *rowBytes* is the number of bytes that are allocated for each scan line in each plane of data. A scan line is a single row of pixels spanning the width of the image.

Normally, *rowBytes* can be figured from the *width* of the image, the number of bits per pixel in each sample (*bps*), and, if the data is in a meshed configuration, the number of samples per pixel (*spp*). However, if the data for each row is aligned on word or other boundaries, it may have been necessary to allocate more memory for each row than there is data to fill it. *rowBytes* lets the object know whether that's the case. If *rowBytes* is 0, the NSBitmapImageRep assumes that there's no empty space at the end of a row.

- *pixelBits* informs the NSBitmapImageRep how many bits are actually allocated per pixel in each plane of data. If the data is in planar configuration, this normally equals *bps* (bits per sample). If the data is in meshed configuration, it normally equals *bps* times *spp* (samples per pixel). However, it's possible for a pixel specification to be followed by some meaningless bits (empty space), as may happen, for example, if pixel data is aligned on byte boundaries. NSBitmapImageRep supports only a limited number of *pixelBits* values (other than the default): for RGB images with 12 *bps*, *pixelBits* may be 16; for RGB images with 24 *bps*, *pixelBits* may be 32. The legal values for *pixelBits* are system dependent.

  If *pixelBits* is 0, the object will interpret the number of bits per pixel to be the expected value, without any meaningless bits.

initWithBitmapHandle:

– (id)**initWithBitmapHandle:**(void *)*bitmap*

On Microsoft Windows platforms, **initWithBitmapHandle:** initializes the receiver, a newly allocated NSBitmapImageRep instance, with the contents of the Windows bitmap indicated by *bitmap*. If **initWithBitmapHandle:** is able to create one or more image representations, it returns **self**. Otherwise, the receiver is freed and **nil** is returned.

initWithData:

– (id)**initWithData:**(NSData *)*bitmapData*

Initializes a newly allocated NSBitmapImageRep from the data found in *bitmapData*. The contents of bitmapData can be any supported bitmap format. For TIFF data, the NSBitmapImageRep is initialized from the first header and image data found in *bitmapData*.

initWithData: returns an initialized NSBitmapImageRep if the initialization was successful, or **nil** if it was unable to interpret the contents of *bitmapData*.

initWithFocusedViewRect:

– (id)**initWithFocusedViewRect:**(NSRect)*rect*

Initializes the receiver, a newly allocated NSBitmapImageRep object, with bitmap data read from a rendered image. The image that's read is located in the current window and is bounded by the *rect* rectangle as specified in the current coordinate system.

This method uses PostScript imaging operators to read the image data into a buffer; the object is then created from that data. The object is initialized with information about the image obtained from the Window Server.

If for any reason the new object can't be initialized, this method frees it and returns **nil**. Otherwise, it returns the initialized object (**self**).

initWithIconHandle:

– (id)**initWithIconHandle:**(void *)*icon*

On Microsoft Windows platforms, **initWithIconHandle:** initializes the receiver, a newly allocated NSBitmapImageRep instance, with the contents of the Windows icon indicated by *icon*. If **initWithIconHandle:** is able to create one or more image representations, it returns **self**. Otherwise, the receiver is freed and **nil** is returned.

isPlanar

– (BOOL)**isPlanar**

Returns YES if image data is segregated into a separate plane for each color and coverage component (planar configuration), and NO if the data is integrated into a single plane (meshed configuration).

See also: – **samplesPerPixel**

numberOfPlanes

– (int)**numberOfPlanes**

Returns the number of separate planes that image data is organized into. This is the number of samples per pixel if the data has a separate plane for each component (**isPlanar** returns YES) and 1 if the data is meshed (**isPlanar** returns NO).

See also: – **samplesPerPixel**, – **hasAlpha** (NSImageRep), – **bitsPerSample** (NSImageRep)

samplesPerPixel

– (int)**samplesPerPixel**

Returns the number of components in the data. It includes both color components and the coverage component, if present.

See also: – **hasAlpha** (NSImageRep), – **bitsPerSample** (NSImageRep)

setCompression:factor:

– (void)**setCompression:**(NSTIFFCompression)*compression*
factor:(float)*factor*

Sets the receiver's compression type and compression factor. *compression* is one of the supported compression types listed in the **getTiffCompressionTypes:count:** class method description. *factor* is a value that is specific to the compression type; many types of compression don't support varying degrees of compression, and thus ignore *factor*. JPEG compression allows a compression factor ranging from 0.0 to 255.0, with 0.0 representing minimal compression.

When an NSBitmapImageRep is created, the instance stores the compression type and factor for the source data. **TIFFRepresentation** and **TIFFRepresentationOfImageRepsInArray:** (class method) try to use the stored compression type and factor. Use this method to change the compression type and factor.

See also: **– canBeCompressedUsing:**

TIFFRepresentation

– (NSData *)**TIFFRepresentation**

Returns a TIFF representation of the image, using the compression that's returned by **getCompression: factor:** (if applicable). This method invokes **TIFFRepresentationUsingCompression:factor:** using the stored compression type and factor retrieved from the initial image data or changed using **setCompression: factor:**. If the stored compression type isn't supported for writing TIFF data (for example, NSTIFFCompressionNEXT), the stored compression is changed to NSTIFFCompressionNone and the compression factor to 0.0 before invoking **TIFFRepresentationUsingCompression:factor:**.

If a problem is encountered during generation of the TIFF, **TIFFRepresentation** raises an NSTIFFException or an NSBadBitmapParametersException.

See also: **+ TIFFRepresentationOfImageRepsInArray:**

TIFFRepresentationUsingCompression:factor:

– (NSData *)**TIFFRepresentationUsingCompression:**(NSTIFFCompression)*comp*
factor:(float)*factor*

Returns a TIFF representation of the image, using the specified compression and factor. If the stored compression type isn't supported for writing TIFF data (for example, NSTIFFCompressionNEXT), the stored compression is changed to NSTIFFCompressionNone and the compression factor to 0.0 before the TIFF representation is generated.

If a problem is encountered during generation of the TIFF, **TIFFRepresentation** raises an NSTIFFException or an NSBadBitmapParametersException.

See also: **– canBeCompressedUsing:**, **+ TIFFRepresentationOfImageRepsInArray:**

NSBox

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

Class Description

An NSBox object is a simple NSView that can do two things: It can draw a border around itself and it can title itself. You can use an NSBox to group, visually, some number of other NSViews. These other NSViews are added to the NSBox through the typical subview-adding methods, such as **addSubview:** and **replaceSubview:with:**.

An NSBox contains a *content area*, a rectangle set within the NSBox's frame in which the NSBox's subviews are displayed. The size and location of the content area depends on the NSBox's border type, title location, the size of the font used to draw the title, and an additional measure that you can set through the **setContentViewMargins:** method. When you create an NSBox, an instance of NSView is created and added (as a subview of the NSBox object) to fill the NSBox's content area. If you replace this *content view* with an NSView of your own, your NSView will be resized to fit the content area. Similarly, as you resize an NSBox its content view is automatically resized to fill the content area.

The NSViews that you add as subviews to an NSBox are actually added to the NSBox's content view—NSView's subview-adding methods are redefined by NSBox to ensure that a subview is correctly placed in the view hierarchy. However, you should note that the **subviews** method *isn't* redefined: It returns an NSArray containing a single object, the NSBox's content view.

Method Types

Getting and modifying the border and title

– borderRect
– borderType
– setBorderType:
– setTitle:
– setTitleFont:
– setTitlePosition:
– setTitleWithMnemonic:
– title
– titleCell
– titleFont
– titlePosition
– titleRect

Setting and placing the content view

– contentView
– contentViewMargins
– setContentView:
– setContentViewMargins:

Resizing the box

– setFrameFromContentFrame:
– sizeToFit

Instance Methods

borderRect

– (NSRect)**borderRect**

Returns the rectangle in which the border is drawn.

borderType

– (NSBorderType)**borderType**

Returns the NSBox's border type. Border types are defined in **NSView.h**; currently, the following border types are defined:

NSNoBorder
NSLineBorder
NSBezelBorder
NSGrooveBorder

By default, an NSBox's border type is NSGrooveBorder.

contentView

– (id)**contentView**

Returns the NSBox's content view. The content view is created automatically when the box is created, and resized as the box is resized (you should never send frame-altering messages directly to a box's content view). You can replace it with an NSView of your own through the **setContentView:** method.

contentViewMargins

– (NSSize)**contentViewMargins**

Returns the distances between the border and the content view. By default, on Mach systems both the width (the horizontal distance between the innermost edge of the border and the content view) and the height (the vertical distance between the innermost edge of the border and the content view) of the returned NSSize are 5.0 in the box's coordinate system.

setBorderType:

– (void)**setBorderType:**(NSBorderType)*aType*

Sets the border type to *aType*, which must be a valid border type. Border types are defined in **NSView.h**; currently, the following border types are defined:

NSNoBorder
NSLineBorder
NSBezelBorder
NSGrooveBorder

If the size of the new border is different from that of the old border, the content view is resized to absorb the difference and the box is marked for redisplay.

See also: **– setNeedsDisplay:** (NSView)

setContentView:

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

Sets the NSBox's content view to *aView*, resizing the NSView to fit within the box's current content area. The box is marked for redisplay.

See also: **– setFrameFromContentFrame:**, **– sizeToFit**, **– setNeedsDisplay:** (NSView)

setContentViewMargins:

– (void)**setContentViewMargins:**(NSSize)*offsetSize*

Sets the horizontal and vertical distance between the border of the NSBox and its content view. The *horizontal* value is applied (reckoned in the box's coordinate system) fully and equally to the left and right sides of the box. The *vertical* value is similarly applied to the top and bottom.

Unlike changing a box's other attributes, such as its title position or border type, changing the offsets *doesn't* automatically resize the content view. In general, you should send a **sizeToFit** message to the box after changing the size of its offsets. This causes the content view to remain unchanged while the box is sized to fit around it.

setFrameFromContentFrame:

– (void)**setFrameFromContentFrame:**(NSRect)*contentFrame*

Places the NSBox so its content view lies on *contentFrame*, reckoned in the coordinate system of the box's superview. The box is marked for redisplay.

See also: – **setContentViewMargins:**, – **setFrame:** (NSView), **– setNeedsDisplay:** (NSView)

setTitle:

– (void)**setTitle:**(NSString *)*aString*

Sets the title to *aString*, and marks the region of the receiver within the title rectangle as needing display. By default, an NSBox's title is "Title". If the size of the new title is different from that of the old title, the content view is resized to absorb the difference.

See also: **– setNeedsDisplayInRect:** (NSView), **– titleRect**

setTitleFont:

– (void)**setTitleFont:**(NSFont *)*aFont*

Sets *aFont* as the NSFont object used to draw the NSBox's title, and marks the region of the receiver within the title rectangle as needing display. On Mach systems the title is drawn using the 12.0 point system font by default. If the size of the new font is different from that of the old font, the content view is resized to absorb the difference.

See also: **– setNeedsDisplayInRect:** (NSView)

setTitlePosition:

– (void)**setTitlePosition:**(NSTitlePosition)*aPosition*

Sets the title position to *aPosition*, which can be one of the values listed in the following table. The default position is NSAtTop.

Value	Meaning
NSNoTitle	The box has no title
NSAboveTop	Title positioned above the box's top border
NSAtTop	Title positioned within the box's top border
NSBelowTop	Title positioned below the box's top border
NSAboveBottom	Title positioned above the box's bottom border
NSAtBottom	Title positioned within the box's bottom border
NSBelowBottom	Title positioned below the box's bottom border

If the new title position changes the size of the box's border area, the content view is resized to absorb the difference, and the box is marked as needing redisplay.

See also: **– setNeedsDisplay:** (NSView)

setTitleWithMnemonic:

– (void)**setTitleWithMnemonic:**(NSString *)*aString*

Sets the title to *aString*, taking into account the fact that an embedded "&" character is not a literal but instead marks the title's "mnemonic." The character immediately following the "&" character will be underlined.

By default, an NSBox's title is "Title". The content view is not automatically resized, and the box is not marked for redisplay.

See also: **– setTitleWithMnemonic:** (NSCell)

sizeToFit

– (void)**sizeToFit**

Resizes and moves the NSBox's content view so that it just encloses its subviews. The box itself is then moved and resized to wrap around the content view. The box's width is constrained so its title will be fully displayed.

You should invoke this method after:

- Adding a subview (to the content view).
- Altering the size or location of such a subview.
- Setting the margins around the content view.

The mechanism by which the content view is moved and resized depends on whether the object responds to its own **sizeToFit** message: If it does respond, then that message is sent and the content view is expected to be so modified. If the content view doesn't respond, the box moves and resizes the content view itself.

title

– (NSString *)**title**

Returns the NSBox's title. By default, a box's title is "Title".

titleCell

– (id)**titleCell**

Returns the NSCell that's used to display the NSBox's title.

titleFont

– (NSFont *)**titleFont**

Returns the NSFont that's used to draw the NSBox's title. On Mach systems the title is drawn using the 12.0 point system font by default.

titlePosition

– (NSTitlePosition)**titlePosition**

Returns a constant representing the title position. See the description of **setTitlePosition:** for a list of the title position constants.

titleRect

– (NSRect)**titleRect**

Returns the rectangle in which the NSBox's title is drawn.

See also: **– setTitlePosition:**, **– setTitle:**, **– setTitleFont:**, **– setFrameFromContentFrame:**, **– sizeToFit**

NSBrowser

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

Class Description

NSBrowser provides a user interface for displaying and selecting items from a list of data, or from hierarchically organized lists of data such as directory paths. When working with a hierarchy of data, the levels are displayed in columns, which are numbered from left to right, beginning with 0. Each column consists of an NSScrollView containing an NSMatrix filled with NSBrowserCells. NSBrowser relies on a delegate to provide the data in its NSBrowserCells. See the NSBrowserCell class description for more on its implementation.

Browser Selection

An entry in an NSBrowser's column can be either a branch node (such as a directory) or a leaf node (such as a file). When the user selects a single branch node entry in a column, the NSBrowser sends itself the **addColumn** message, which messages its delegate to load the next column. The user's selection can be represented as a character string; if the selection is hierarchical (for example, a filename within a directory), each component of the path to the selected node is separated by "/". To use some other character as the delimiter, invoke **setPathSeparator:**.

An NSBrowser can be set to allow selection of multiple entries in a column, or to limit selection to a single entry. When set for multiple selection, it can also be set to limit multiple selection to leaf nodes only, or to allow selection of both types of nodes together.

As a subclass of NSControl, NSBrowser has a target object and action message. Each time the user selects one or more entries in a column, the action message is sent to the target. NSBrowser also adds an action to be sent when the user double-clicks on an entry, which allows the user to select items without any action being taken, and then double-click to invoke some useful action such as opening a file.

User Interface Features

The user interface features of an NSBrowser can be changed in a number of ways. The NSBrowser may or may not have a horizontal scroller. (The NSBrowser's columns, by contrast, always have vertical scrollers—although a scroller's buttons and knob might be invisible if the column doesn't contain many

entries.) You generally shouldn't create an NSBrowser without a horizontal scroller; if you do, you must make sure the bounds rectangle of the NSBrowser is wide enough that all the columns can be displayed. An NSBrowser's columns may be bordered and titled, bordered and untitled, or unbordered and untitled. A column's title may be taken from the selected entry in the column to its left, or may be provided explicitly by the NSBrowser or its delegate.

NSBrowser's Delegate

NSBrowser requires a delegate to provide it with data to display. The delegate is responsible for providing the data and for setting each item as a branch or leaf node, enabled or disabled. It can also receive notification of events like scrolling and requests for validation of columns that may have changed.

You can implement one of two delegate types: active or passive. An active delegate creates a column's rows (that is, the NSBrowserCells) itself, while a passive one leaves that job to the NSBrowser. Normally, passive delegates are preferable, because they're easier to implement. An active delegate must implement **browser:createRowsForColumn:inMatrix:** to create the rows of the specified column. A passive delegate, on the other hand, must implement **browser:numberOfRowsInColumn:** to let the NSBrowser know how many rows to create. These two methods are mutually exclusive; you can implement one or the other, but not both. (The NSBrowser ascertains what type of delegate it has by which method the delegate responds to.)

Both types of delegate implement **browser:willDisplayCell:atRow:column:** to set up state (such as the cell's string value and whether the cell is a leaf or a branch) before an individual cell is displayed. (This delegate method doesn't need to invoke NSBrowserCell's **setLoaded:** method, because the NSBrowser can determine that state by itself.) An active delegate can instead set all the cells' state at the time the cells are created, in which case it doesn't need to implement **browser:willDisplayCell:atRow:column:**. However, a passive delegate must always implement this method.

Method Types

Setting component classes

+ cellClass
– setCellClass:
– cellPrototype
– setCellPrototype:
– matrixClass
– setMatrixClass:

Getting matrices, cells, and rows

– selectedCell
– selectedCellInColumn:
– selectedCells
– selectAll:
– selectedRowInColumn:
– selectRow:inColumn:
– loadedCellAtRow:column:
– matrixInColumn:

Getting and setting paths

– path
– setPath:
– pathToColumn:
– pathSeparator
– setPathSeparator:

Manipulating columns

– addColumn
– displayAllColumns
– displayColumn:
– columnOfMatrix:
– selectedColumn
– lastColumn
– setLastColumn:
– firstVisibleColumn
– numberOfVisibleColumns
– lastVisibleColumn
– validateVisibleColumns

Loading columns

– isLoaded
– loadColumnZero
– reloadColumn:

Setting selection characteristics

– allowsBranchSelection
– setAllowsBranchSelection:
– allowsEmptySelection
– setAllowsEmptySelection:
– allowsMultipleSelection
– setAllowsMultipleSelection:

Setting column characteristics

– reusesColumns
– setReusesColumns:
– maxVisibleColumns
– setMaxVisibleColumns:
– minColumnWidth
– setMinColumnWidth:
– separatesColumns
– setSeparatesColumns:
– takesTitleFromPreviousColumn
– setTakesTitleFromPreviousColumn:

Manipulating column titles

– titleOfColumn:
– setTitle:ofColumn:
– isTitled
– setTitled:
– drawTitleOfColumn:inRect:
– titleHeight
– titleFrameOfColumn:

Scrolling an NSBrowser

– scrollColumnToVisible:
– scrollColumnsLeftBy:
– scrollColumnsRightBy:
– updateScroller
– scrollViaScroller:

Showing a horizontal scroller

– hasHorizontalScroller
– setHasHorizontalScroller:

Setting the behavior of arrow keys

– acceptsArrowKeys
– setAcceptsArrowKeys:
– sendsActionOnArrowKeys
– setSendsActionOnArrowKeys:

Getting column frames

– frameOfColumn:
– frameOfInsideOfColumn:

Arranging browser components

– tile

Setting the delegate

– delegate
– setDelegate:

Target and action

– doubleAction
– setDoubleAction:
– sendAction

Event handling

– doClick:
– doDoubleClick:

Class Methods

cellClass

+ (Class)**cellClass**

Returns the NSBrowserCell class (regardless of whether a **setCellClass:** message has been sent to a particular instance)

See also: – **cellPrototype**,– s**etCellPrototype:**

Instance Methods

acceptsArrowKeys

– (BOOL)**acceptsArrowKeys**

Returns YES if the arrow keys are enabled.

See also: – **setAcceptsArrowKeys:**

addColumn

– (void)**addColumn**

Adds a column to the right of the last column.

See also: – **columnOfMatrix:**, – **displayColumn:**, – **selectedColumn**

allowsBranchSelection

– (BOOL)**allowsBranchSelection**

Returns whether the user can select branch items when multiple selection is enabled.

See also: – **setAllowsBranchSelection:**

allowsEmptySelection

– (BOOL)**allowsEmptySelection**

Returns whether there can be nothing selected.

See also: – **setAllowsEmptySelection:**

allowsMultipleSelection

– (BOOL)**allowsMultipleSelection**

Returns whether the user can select multiple items.

See also: – **setAllowsMultipleSelection:**

cellPrototype

– (id)**cellPrototype**

Returns the NSBrowser's prototype NSCell.

See also: – **setCellPrototype:**, – **setCellClass:**

columnOfMatrix:

– (int)**columnOfMatrix:**(NSMatrix *)*matrix*

Returns the column number in which *matrix* is located.

See also: – **matrixInColumn:**

delegate

– (id)**delegate**

Returns the NSBrowser's delegate.

See also: – **setDelegate:**

displayAllColumns

– (void)**displayAllColumns**

Updates the NSBrowser to display all loaded columns.

See also: – **addColumn**, – **validateVisibleColumns**

displayColumn:

– (void)**displayColumn:**(int)*column*

Updates the NSBrowser to display the column with the given index.

See also: – **addColumn**, – **validateVisibleColumns**

doClick:

– (void)**doClick:**(id)*sender*

Responds to (single) mouse clicks in a column of the NSBrowser.

See also: – **sendAction**

doDoubleClick:

– (void)**doDoubleClick:**(id)*sender*

Responds to double-clicks in a column of the NSBrowser.

See also: – **setDoubleAction:**

doubleAction

– (SEL)**doubleAction**

Returns the NSBrowser's double-click action method.

See also: – **setDoubleAction:**

drawTitleOfColumn:inRect:

– (void)**drawTitleOfColumn:**(int)*column* **inRect:**(NSRect)*aRect*

Draws the title for the column at index *column* within the rectangle defined by *aRect*.

See also: – **setTitle:ofColumn:**, – **titleFrameOfColumn:**, – **titleHeight**

firstVisibleColumn

– (int)**firstVisibleColumn**

Returns the index of the first visible column.

See also: – **lastVisibleColumn**, – **numberOfVisibleColumns**

frameOfColumn:

– (NSRect)**frameOfColumn:**(int)*column*

Returns the rectangle containing the column at index column.

frameOfInsideOfColumn:

– (NSRect)**frameOfInsideOfColumn:**(int)*column*

Returns the rectangle containing the column at index column, not including borders.

hasHorizontalScroller

– (BOOL)**hasHorizontalScroller**

Returns whether an NSScroller is used to scroll horizontally.

See also: – **setHasHorizontalScroller:**

isLoaded

– (BOOL)**isLoaded**

Returns whether column zero is loaded.

See also: – **loadColumnZero**, – **reloadColumn:**

isTitled

– (BOOL)**isTitled**

Returns whether columns display titles.

See also: – **setTitled:**

lastColumn

– (int)**lastColumn**

Returns the index of the last column loaded.

See also: – **selectedColumn**, – **setLastColumn:**

lastVisibleColumn

– (int)**lastVisibleColumn**

Returns the index of the last visible column.

See also: – **firstVisibleColumn**, – **numberOfVisibleColumns**

loadColumnZero

– (void)**loadColumnZero**

Loads column zero; unloads previously loaded columns.

See also: – **isLoaded**, – **reloadColumn:**

loadedCellAtRow:column:

– (id)**loadedCellAtRow:**(int)*row* **column:**(int)*column*

Loads if necessary and returns the NSCell at *row* in *column*.

See also: – **selectedCellInColumn:**

matrixClass

– (Class)**matrixClass**

Returns the class of NSMatrix used in the NSBrowser's columns.

See also: – **setMatrixClass:**

matrixInColumn:

– (NSMatrix *)**matrixInColumn:**(int)*column*

Returns the matrix located in the column identified by index *column*.

maxVisibleColumns

– (int)**maxVisibleColumns**

Returns the maximum number of visible columns.

See also: – **setMaxVisibleColumns:**

minColumnWidth

– (float)**minColumnWidth**

Returns the minimum column width in pixels.

See also: – **setMinColumnWidth:**

numberOfVisibleColumns

– (int)**numberOfVisibleColumns**

Returns the number of columns visible.

See also: – **validateVisibleColumns**

path

– (NSString *)**path**

Returns the browser's current path.

See also: – **setPath:**

pathSeparator

– (NSString *)**pathSeparator**

Returns the path separator. The default is “/”.

See also: – **setPathSeparator:**

pathToColumn:

– (NSString *)**pathToColumn:**(int)*column*

Returns a string representing the path from the first column up to, but not including, the column at index *column*.

See also: – **path**, – **setPath:**

reloadColumn:

– (void)**reloadColumn:**(int)*column*

Reloads *column* if it is loaded; sets it as the last column.

See also: – **isLoaded**, – **loadColumnZero**

reusesColumns

– (BOOL)**reusesColumns**

Returns YES if NSMatrix objects aren’t freed when their columns are unloaded.

See also: – **setReusesColumns:**

scrollColumnToVisible:

– (void)**scrollColumnToVisible:**(int)*column*

Scrolls to make the column at index column visible.

See also: – **scrollViaScroller:**, – **updateScroller**

scrollColumnsLeftBy:

– (void)**scrollColumnsLeftBy:**(int)*shiftAmount*

Scrolls columns left by shiftAmount columns.

See also: – **scrollViaScroller:**, – **updateScroller**

scrollColumnsRightBy:

– (void)**scrollColumnsRightBy:**(int)*shiftAmount*

Scrolls columns right by shiftAmount columns.

See also: – **scrollViaScroller:**, – **updateScroller**

scrollViaScroller:

– (void)**scrollViaScroller:**(NSScroller *)*sender*

Scrolls columns left or right based on an NSScroller.

See also: – **updateScroller**

selectAll:

– (void)**selectAll:**(id)*sender*

Selects all NSCells in the last column of the NSBrowser.

See also: – **selectedCell**, – **selectedCells**, – **selectedColumn**

selectRow:inColumn:

– (void)**selectRow:**(int)*row* **inColumn:**(int)*column*

Selects the cell at index *row* in the column identified by index *column*.

See also: – **loadedCellAtRow:column:**, – **selectedRowInColumn:**

selectedCell

– (id)**selectedCell**

Returns the last (rightmost and lowest) selected NSCell.

See also: – **loadedCellAtRow:column:**, – **selectedCell**s, – **selectRow:inColumn:**

selectedCellInColumn:

– (id)**selectedCellInColumn:**(int)*column*

Returns the last (lowest) NSCell that's selected in column.

See also: – **loadedCellAtRow:column:**, – **selectedCell**, – **selectedRowInColumn:**

selectedCells

– (NSArray *)**selectedCells**

Returns all cells selected in the rightmost column.

See also: – **selectAll:**, – **selectedCell**

selectedColumn

– (int)**selectedColumn**

Returns the index of the last column with a selected item.

See also: – **columnOfMatrix:**, – **selectAll:**

selectedRowInColumn:

– (int)**selectedRowInColumn:**(int)*column*

Returns the row index of the selected cell in the column specified by index *column*.

See also: – **loadedCellAtRow:column:**, – **selectedCell**, – **selectedCellInColumn:**

sendAction

– (BOOL)**sendAction**

Sends the action message to the target. Returns YES upon success, NO if no target for the message could be found.

sendsActionOnArrowKeys

– (BOOL)**sendsActionOnArrowKeys**

Returns NO if pressing an arrow key only scrolls the browser, YES if it also sends the action message specified by **setAction:**.

See also: – **acceptsArrowKeys**, – **setSendsActionOnArrowKeys:**

separatesColumns

– (BOOL)**separatesColumns**

Returns whether columns are separated by bezeled borders.

See also: – **setSeparatesColumns:**

setAcceptsArrowKeys:

– (void)**setAcceptsArrowKeys:**(BOOL)*flag*

Enables or disables the arrow keys as used for navigating within and between browsers.

See also: – **acceptsArrowKeys**, – **sendsActionOnArrowKeys**

setAllowsBranchSelection:

– (void)**setAllowsBranchSelection:**(BOOL)*flag*

Sets whether the user can select branch items when multiple selection is enabled.

See also: – **allowsBranchSelection**

setAllowsEmptySelection:

– (void)**setAllowsEmptySelection:**(BOOL)*flag*

Sets whether there can be nothing selected.

See also: – **allowsEmptySelection**

setAllowsMultipleSelection:

– (void)**setAllowsMultipleSelection:**(BOOL)*flag*

Sets whether the user can select multiple items.

See also: – **allowsMultipleSelection**

setCellClass:

– (void)**setCellClass:**(Class)*factoryId*

Sets the class of NSCell used in the columns of the NSBrowser.

See also: + **cellClass**, – **cellPrototype**

setCellPrototype:

– (void)**setCellPrototype:**(NSCell *)*aCell*

Sets the NSCell instance copied to display items in the columns of NSBrowser.

See also: + **cellClass**, – **cellPrototype**

setDelegate:

– (void)**setDelegate:**(id)*anObject*

Sets the NSBrowser's delegate to *anObject*. Raises NSBrowserIllegalDelegateException if the delegate specified by *anObject* doesn't respond to **browser:willDisplayCell:atRow:column:** and either of the methods **browser:numberOfRowsInColumn:** or **browser:createRowsForColumn:inMatrix:**.

See also: – **delegate**

setDoubleAction:

– (void)**setDoubleAction:**(SEL)*aSelector*

Sets the NSBrowser's double-click action to *aSelector*.

See also: – **doubleAction**, – **sendAction**

setHasHorizontalScroller:

– (void)**setHasHorizontalScroller:**(BOOL)*flag*

Sets whether an NSScroller is used to scroll horizontally.

See also: – **hasHorizontalScroller**

setLastColumn:

– (void)**setLastColumn:**(int)*column*

Sets the last column to column.

See also: – **lastColumn**, – **lastVisibleColumn**

setMatrixClass:

– (void)**setMatrixClass:**(Class)*factoryId*

Sets the matrix class (NSMatrix or an NSMatrix subclass) used in the NSBrowser's columns.

See also: – **matrixClass**

setMaxVisibleColumns:

– (void)**setMaxVisibleColumns:**(int)*columnCount*

Sets the maximum number of columns displayed.

See also: – **maxVisibleColumns**

setMinColumnWidth:

– (void)**setMinColumnWidth:**(float)*columnWidth*

Sets the minimum column width in pixels.

See also: – **minColumnWidth**

setPath:

– (BOOL)**setPath:**(NSString *)*path*

Parses *path* and selects corresponding items in the NSBrowser columns.

See also: – **path**, – **pathToColumn:**

setPathSeparator:

– (void)**setPathSeparator:**(NSString *)*newString*

Sets the path separator to *newString*.

See also: – **pathSeparator**

setReusesColumns:

– (void)**setReusesColumns:**(BOOL)*flag*

If *flag* is YES, prevents NSMatrix objects from being freed when their columns are unloaded, so they can be reused.

See also: – **reusesColumns**

setSendsActionOnArrowKeys:

– (void)**setSendsActionOnArrowKeys:**(BOOL)*flag*

Sets whether pressing an arrow key will cause the action message to be sent (in addition to causing scrolling).

See also: – **sendsActionOnArrowKeys**

setSeparatesColumns:

– (void)**setSeparatesColumns:**(BOOL)*flag*

Sets whether to separate columns with bezeled borders.

See also: – **separatesColumns**

setTakesTitleFromPreviousColumn:

– (void)**setTakesTitleFromPreviousColumn:**(BOOL)*flag*

Sets whether the title of a column is set to the string value of the selected NSCell in the previous column.

See also: – **takesTitleFromPreviousColumn**

setTitle:ofColumn:

– (void)**setTitle:**(NSString *)*aString* **ofColumn:**(int)*column*

Sets the title of the column at index *column* to *aString*.

See also: – **drawTitleOfColumn:inRect:**, – **titleOfColumn:**

setTitled:

– (void)**setTitled:**(BOOL)*flag*

Sets whether columns display titles.

See also: – **isTitled**

takesTitleFromPreviousColumn

– (BOOL)**takesTitleFromPreviousColumn**

Returns YES if the title of a column is set to the string value of the selected NSCell in the previous column.

See also: – **setTakesTitleFromPreviousColumn:**

tile

– (void)**tile**

Adjusts the various subviews of NSBrowser—scrollers, columns, titles, and so on—without redrawing. Your code shouldn't send this message. It's invoked any time the appearance of the NSBrowser changes.

titleFrameOfColumn:

– (NSRect)**titleFrameOfColumn:**(int)*column*

Returns the bounds of the title frame for the column at index *column*.

See also: – **drawTitleOfColumn:inRect:**

titleHeight

– (float)**titleHeight**

Returns the height of column titles.

See also: – **drawTitleOfColumn:inRect:**

titleOfColumn:

– (NSString *)**titleOfColumn:**(int)*column*

Returns the title displayed for the column at index *column*.

See also: – **setTitle:ofColumn:**

updateScroller

– (void)**updateScroller**

Updates the horizontal scroller to reflect column positions.

See also: – **scrollViaScroller:**

validateVisibleColumns

– (void)**validateVisibleColumns**

Invokes delegate method **browser:isColumnValid:** for visible columns.

See also: – **numberOfVisibleColumns**

Methods Implemented By the Delegate

browser:createRowsForColumn:inMatrix:

– (void)**browser:**(NSBrowser *)*sender*
createRowsForColumn:(int)*column*
inMatrix:(NSMatrix *)*matrix*

Creates a row in matrix for each row of data to be displayed in column of the browser. Either this method or **browser:numberOfRowsInColumn:** must be implemented, but not both (or an NSBrowserIllegalDelegateException will be raised).

See also: – **browser:willDisplayCell:atRow:column:**

browser:isColumnValid:

– (BOOL)**browser:**(NSBrowser *)*sender* **isColumnValid:**(int)*column*

Returns whether the contents of the specified column are valid. If NO is returned, *sender* reloads the column. This method is invoked in response to **validateVisibleColumns** being sent to *sender*.

browser:numberOfRowsInColumn:

– (int)**browser:**(NSBrowser *)*sender* **numberOfRowsInColumn:**(int)*column*

Returns the number of rows of data in the column at index *column*. Either this method or **browser: createRowsForColumn:inMatrix:** must be implemented, but not both.

See also: – **browser:willDisplayCell:atRow:column:**

browser:selectCellWithString:inColumn:

– (BOOL)**browser:**(NSBrowser *)*sender*
selectCellWithString:(NSString *)*title*
inColumn:(int)*column*

Asks the delegate to select the NSCell with title *title* in the column at index *column*. If the delegate returns NO, the NSCell is not selected.

See also: – **selectedCellInColumn:**

browser:selectRow:inColumn:

– (BOOL)**browser:**(NSBrowser *)*sender*
selectRow:(int)*row*
inColumn:(int)*column*

Asks the delegate to select the NSCell at row *row* in the column at index *column*. If the delegate returns NO, the NSCell is not selected.

See also: – **selectedRowInColumn:**, – **selectRow:inColumn:**

browser:titleOfColumn:

– (NSString *)**browser:**(NSBrowser *)*sender* **titleOfColumn:**(int)*column*

Asks the delegate for the title to display above the column at index *column*.

See also: – **setTitle:ofColumn:**, – **titleOfColumn:**

browser:willDisplayCell:atRow:column:

– (void)**browser:**(NSBrowser *)*sender*
willDisplayCell:(id)*cell*
atRow:(int)*row*
column:(int)*column*

Notifies the delegate before the NSBrowser displays the specified *cell* at *row* in *column*. The delegate should set any state necessary for the correct display of the cell.

See also: **– browser:createRowsForColumn:inMatrix:, – browser:numberOfRowsInColumn:**

browserDidScroll:

– (void)**browserDidScroll:**(NSBrowser *)*sender*

Notifies the delegate when the NSBrowser has scrolled.

browserWillScroll:

– (void)**browserWillScroll:**(NSBrowser *)*sender*

Notifies the delegate when the NSBrowser will scroll.

NSBrowserCell

Inherits From:	NSCell : NSObject
Conforms To:	NSCoding, NSCopying (NSCell) NSObject (NSObject)
Declared In:	AppKit/NSBrowserCell.h

Class Description

NSBrowserCell is the subclass of NSCell used by default to display data in the columns of an NSBrowser. (Each column contains an NSMatrix filled with NSBrowserCells.) An NSBrowserCell can be a leaf or branch cell. A branch cell displays an image indicating that, when the cell is clicked, the NSBrowser will display a new column of NSBrowserCells; branch cells are thus important to the display of hierarchical information typical of NSBrowsers. An NSBrowserCell can also be loaded or unloaded; loaded NSBrowserCells have their state set and are ready for display.

Many of NSBrowserCell's methods are designed to interact with NSBrowser and NSBrowser's delegate. The delegate implements methods for loading the NSCells in NSBrowser by setting their values and status. If your code needs access to a specific NSBrowserCell, you can use the NSBrowser method **loadedCellAtRow:column:**.

Because NSBrowserCells do not inherit from NSActionCell, they don't hold target and action values and thus don't participate in the target/action paradigm of the Application Kit. However, NSBrowser does allow you to specify a target and an action, and you can obtain the last selected NSBrowserCell by sending **selectedCell** to an NSBrowser.

You may find it useful to create a subclass of NSBrowserCell to alter its behavior and to enable it to work with and display the type of data you wish to represent. Use NSBrowser's **setCellClass:** or **setCellPrototype:** methods to have it use your subclass.

See the NSBrowser class specification for more details. In particular, the class description and the "Methods Implemented by the Delegate" section describe how the NSBrowser's delegate interacts with both NSBrowser and NSBrowserCells.

Method Types

Accessing graphic images

+ branchImage
+ highlightedBranchImage
– alternateImage
– setAlternateImage:

Setting state

– reset
– set

Determining cell attributes

– isLeaf
– setLeaf:
– isLoaded
– setLoaded:

Class Methods

branchImage

+ (NSImage *)**branchImage**

Returns the default image for branch NSBrowserCells (a right-pointing triangle). Override this method if you want a different image. To have a branch NSBrowserCell with no image (and no space reserved for an image), override this method to return **nil**.

See also: – **alternateImage**, + **highlightedBranchImage**, – **setAlternateImage:**

highlightedBranchImage

+ (NSImage *)**highlightedBranchImage**

Returns the default NSImage for branch NSBrowserCells that are highlighted (a lighter version of the image returned by **branchImage**). Override this method if you want a different image.

See also: + **branchImage**, – **alternateImage**, – **setAlternateImage:**

Instance Methods

alternateImage

– (NSImage *)**alternateImage**

Returns this NSBrowserCell's image for the highlighted state or **nil** if no image is set.

See also: – **setAlternateImage:**

isLeaf

– (BOOL)**isLeaf**

Returns whether the NSBrowserCell is a leaf or a branch cell. A branch NSBrowserCell has an image near its right edge indicating that more, hierarchically related information is available; when the user selects the cell, the NSBrowser displays a new column of NSBrowserCells. A leaf NSBrowserCell has no image, indicating that the user has reached a terminal piece of information; it doesn't point to additional information.

See also: – **setLeaf:**

isLoaded

– (BOOL)**isLoaded**

Returns YES if all the NSBrowserCell's state has been set and the cell is ready to display.

See also: – **setLoaded:**

reset

– (void)**reset**

Unhighlights the NSBrowserCell and sets its state to 0 (NO).

See also: – **set**

set

– (void)**set**

Highlights the NSBrowserCell and sets its state to 1 (YES).

See also: – **reset**

setAlternateImage:

– (void)**setAlternateImage:**(NSImage *)*newAltImage*

Sets this NSBrowserCell's image for the highlighted state, retaining the image. If *newAltImage* is **nil**, it removes the alternate image for the NSBrowserCell.

See also: – **alternateImage**

setLeaf:

– (void)**setLeaf:**(BOOL)*flag*

Sets whether the NSBrowserCell is a leaf or a branch cell. A branch NSBrowserCell has an image near its right edge indicating that more, hierarchically related information is available; when the user selects the cell, the NSBrowser displays a new column of NSBrowserCells. A leaf NSBrowserCell has no image, indicating that the user has reached a terminal piece of information; it doesn't point to additional information.

See also: – **isLeaf**

setLoaded:

– (void)**setLoaded:**(BOOL)*flag*

Sets whether the NSBrowserCell's state has been set to 1 (YES) and the cell is ready to display.

See also: – **isLoaded**

NSBundle Additions

Inherits From: NSObject

Declared In: AppKit/NSHelpManager.h
AppKit/NSImage.h
AppKit/NSNibLoading.h

Class Description

The Application Kit adds methods to the Foundation Framework's NSBundle class for:

- Loading nib files
- Locating image resources
- Accessing context help from a **Help.plist** file

These methods become part of the NSBundle class only for those applications that use the Application Kit.

For information on bundles, see the NSBundle class specification in the *Foundation Framework Reference*.

Method Types

Loading nib files

+ loadNibFile:externalNameTable:withZone:
+ loadNibNamed:owner:
– loadNibFile:externalNameTable:withZone:

Locating NSImage resources

– pathForImageResource:

Accessing context help

– contextHelpForKey:

Class Methods

loadNibFile:externalNameTable:withZone:

+ (BOOL)**loadNibFile:**(NSString *)*fileName*
externalNameTable:(NSDictionary *)*context*
withZone:(NSZone *)*zone*

Unarchives the contents of the nib file whose absolute path is *fileName*. Objects from the nib file are allocated in the memory zone specified by *zone*. The *context* argument is a name table—a dictionary whose keys are names like "NSOwner" and whose values are existing objects that can be referenced by the newly unarchived objects. Returns YES upon success, or NO if the specified nib file couldn't be loaded.

This method is declared in **NSNibLoading.h**.

loadNibNamed:owner:

+ (BOOL)**loadNibNamed:**(NSString *)*aNibName* **owner:**(id)*owner*

Similar to **loadNibFile:externalNameTable:withZone:**, but the name table's only element is the object specified by *owner* (stored with the key "NSOwner"). Objects from the nib file are allocated in *owner*'s zone. If there's a bundle for *owner*'s class, this method looks in that bundle for the nib file named *aNibName* (this argument need not include the ".nib" extension); otherwise, it looks in the main bundle.

This method is declared in **NSNibLoading.h**.

See also: **+ bundleForClass:** (NSBundle)

Instance Methods

contextHelpForKey:

– (NSAttributedString *)**contextHelpForKey:**(NSString *)*key*

Returns the context-sensitive help from the help file named *key*; or **nil** if **Help.plist** isn't present or if **Help.plist** doesn't contain an entry for *key.*

When you build your application, **/usr/bin/compileHelp** packages your help files into a property list named **Help.plist**. **contextHelpForKey:** extracts context help from this file, but looks it up using the name of the original help file. For example, if your application project contains a help file **Copy.rtf**, you can get its text using **contextHelpForKey:** with the argument @"Copy.rtf".

This method is declared in **NSHelpManager.h**.

See also: **– contextHelpForObject:** (NSHelpManager)

loadNibFile:externalNameTable:withZone:

– (BOOL)**loadNibFile:**(NSString *)*fileName*
externalNameTable:(NSDictionary *)*context*
withZone:(NSZone *)*zone*

Unarchives the contents of the nib file named *fileName*. The method first looks for the nib file in the language-specific ".lproj" directory; if the nib file isn't there, it looks for a non-localized resource in the immediate bundle directory. Objects from the nib file are allocated in the memory zone specified by *zone*. The *context* argument is a name table—a dictionary whose keys are names like "NSOwner" and whose values are existing objects that can be referenced by the newly unarchived objects. Returns YES upon success, or NO if the specified nib file couldn't be loaded.

This method is declared in **NSNibLoading.h**.

pathForImageResource:

– (NSString *)**pathForImageResource:**(NSString *)*name*

Returns the absolute pathname of the file containing the specified image resource, or **nil** if the specified resource can't be located. Image resources are those files in the bundle which are recognized by NSImage without filtering (essentially, a file whose type is one of those returned by the **imageUnfilteredFileTypes** method). The resource *name* is simply the filename without the path of its bundle directory; the filename extension is optional.

This method is declared in **NSImage.h**.

See also: **– pathForResource:ofType:** (NSBundle)

NSButton

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

Class Description

NSButton is a subclass of NSControl that intercepts mouse-down events and sends an action message to a target object when it's clicked or pressed.

The NSButton can send its action continuously and display highlighting in several different ways. What's more, an NSButton can have a key equivalent that's eligible for triggering whenever the NSButton's NSPanel or NSWindow is the key window.

NSButton and NSMatrix both provide a control view, which is needed to display an NSButtonCell object. However, while NSMatrix requires you to access the NSButtonCells directly, most of NSButton's methods are "covers" for identically declared methods in NSButtonCell. (In other words, the implementation of the NSButton method invokes the corresponding NSButtonCell method for you, allowing you to be unconcerned with the NSButtonCell's existence.) The only NSButtonCell methods that don't have covers relate to the font used to display the key equivalent, and to specific methods for highlighting or showing the NSButton's state (these last are usually set together with NSButton's **setButtonType:** method).

Button States

By virtue of its NSButtonCell, NSButton is an NSControl and displays its state depending on the configuration of the NSButtonCell. The NSButton can have two or three states. If it has two, they are on and off. If it has three, they are on, off, and mixed. A mixed state is useful for a checkbox or radio button that reflects the status of a feature. For example, suppose you have a checkbox that makes the selected text bold. If all the selected text is bold, it's on. If none of the selected text is bold, it's off. If the text has a combination of bold and plain text, it's mixed. Now suppose you click the checkbox. If you turn it on, all the text becomes bold. If you turn it off, all the text becomes plain. If you select the mixed state, the text remains as it is.

By default, a button has two states. You can allow the third state with the method **setAllowsMixedState:**. To set the button's state directly, use **setState:**. To cycle through all available states, use **setNextState**. Note that the state is used as the value, so NSControl methods like **setIntValue:** actually set the state.

Creating a Subclass of NSButton

Override the designated initializer (NSView's **initWithFrame:** method) if you create a subclass of NSButton that performs its own initialization. If you want to use a custom NSButtonCell subclass with your subclass of NSButton, you have to override the **setCellClass:** method, as described in "Creating New NSControls" in the NSControl class specification.

See the NSButtonCell class specification for more on NSButton's behavior.

Method Types

Setting the button type

– setButtonType:

Setting the state

– allowsMixedState
– setAllowsMixedState:
– setNextState
– setState:
– state

Setting the repeat interval

– getPeriodicDelay:interval:
– setPeriodicDelay:interval:

Setting the titles

– alternateTitle
– attributedAlternateTitle
– attributedTitle
– setAlternateTitle:
– setAttributedAlternateTitle:
– setAttributedTitle:
– setTitle:
– setTitleWithMnemonic:
– title

Setting the images

– alternateImage
– image
– imagePosition
– setAlternateImage:
– setImage:
– setImagePosition:

Modifying graphic attributes

– bezelStyle
– isBordered
– isTransparent
– setBordered:
– setBezelStyle:
– setTransparent:

Displaying

– highlight:

Setting the key equivalent

– keyEquivalent
– keyEquivalentModifierMask
– setKeyEquivalent:
– setKeyEquivalentModifierMask:

Handling events and action messages

– performClick:
– performKeyEquivalent:

Instance Methods

allowsMixedState

– (BOOL)**allowsMixedState**

Returns YES if the button has three states: on, off, and mixed. Returns NO if the button has two states: on and off.

See also: – **setAllowsMixedState:**, – **setNextState**

alternateImage

– (NSImage *)**alternateImage**

Returns the image that appears on the button when it's in its alternate state, or **nil** if there is no alternate image. Note that some button types don't display an alternate image. Buttons don't display images by default.

See also: **– image**, **– imagePosition**, **– keyEquivalent**, **– setButtonType:**

alternateTitle

– (NSString *)**alternateTitle**

Returns the string that appears on the button when it's in its alternate state, or the empty string if the button doesn't display an alternate title. Note that some button types don't display an alternate title. By default, a button's alternate title is "Button".

See also: **– attributedAlternateTitle**, **– setButtonType:**, **– title**

attributedAlternateTitle

– (NSAttributedString *)**attributedAlternateTitle**

Returns the string that appears on the button when it's in its alternate state as an NSAttributedString, or an empty attributed string if the button doesn't display an alternate title. Note that some button types don't display an alternate title. By default, a button's alternate title is "Button".

See also: **– setButtonType:**, **– attributedTitle**

attributedTitle

– (NSAttributedString *)**attributedTitle**

Returns the string that appears on the button when it's in its normal state as an NSAttributedString, or an empty attributed string if the button doesn't display a title. A button's title is always displayed if the button doesn't use its alternate contents for highlighting or displaying the alternate state. By default, a button's title is "Button".

See also: **– attributedAlternateTitle**, **– setButtonType:**

bezelStyle

– (NSBezelStyle)**bezelStyle**

Returns the appearance of the button's border. See **setBezelStyle:** for the list of the possible values.

See also: **– setBezelStyle:**

getPeriodicDelay:interval:

– (void)**getPeriodicDelay:**(float *)*delay* **interval:**(float *)*interval*

Returns by reference the delay and interval periods for a continuous button. *delay* is the amount of time (in seconds) that the button will pause before starting to periodically send action messages to the target object. *interval* is the amount of time (also in seconds) between those messages.

Default delay and interval values are taken from a user's defaults (60 seconds maximum for each); if the user hasn't specified default values, *delay* defaults to 0.4 seconds and *interval* defaults to 0.075 seconds.

See also: – **isContinuous** (NSControl)

highlight:

– (void)**highlight:**(BOOL)*flag*

Highlights (or unhighlights) the button according to *flag*. Highlighting may involve the button appearing "pushed in" to the screen, displaying its alternate title or image, or causing the button to appear to be "lit." If the current state of the button matches *flag*, no action is taken.

See also: **– setButtonType:**

image

– (NSImage *)**image**

Returns the image that appears on the button when it's in its normal state, or **nil** if there is no such image. This image is always displayed on a button that doesn't change its contents when highlighting or showing its alternate state. Buttons don't display images by default.

See also: – **alternateImage**, – **setButtonType:**

imagePosition

– (NSCellImagePosition)**imagePosition**

Returns the position of the button's image relative to its title. The return value is one of the following (these are defined in **NSCell.h**):

Return Value	Meaning
NSNoImage	The button doesn't display an image (this is the default)
NSImageOnly	The button displays an image, but not a title
NSImageLeft	The image is to the left of the title
NSImageRight	The image is to the right of the title
NSImageBelow	The image is below the title
NSImageAbove	The image is above the title

Return Value	Meaning
NSImageOverlaps	The image overlaps the title

If the title is above, below, or overlapping the image, or if there is no image, the text is horizontally centered within the button.

See also: – **setButtonType:**, **– setImage:**, **– setTitle:**

isBordered

– (BOOL)**isBordered**

Returns YES if the button has a border, NO otherwise. A button's border isn't the single line of most other controls' borders; instead, it's a raised bezel. By default, buttons are bordered.

isTransparent

– (BOOL)**isTransparent**

Returns YES if the button is transparent, NO otherwise. A transparent button never draws itself, but it receives mouse-down events and tracks the mouse properly.

keyEquivalent

– (NSString *)**keyEquivalent**

Returns the key-equivalent character of the button, or the empty string if one hasn't been defined. Buttons don't have a default key equivalent.

See also: **– keyEquivalentFont** (NSButtonCell), – **performKeyEquivalent:**

keyEquivalentModifierMask

– (unsigned int)**keyEquivalentModifierMask**

Returns the mask indicating the modifier keys that are applied to the button's key equivalent. Mask bits are defined in **NSEvent.h**; only NSControlKeyMask, NSAlternateKeyMask, and NSCommandKeyMask bits are relevant in button key-equivalent modifier masks.

See also: **– keyEquivalent**

performClick:

– (void)**performClick:**(id)*sender*

Simulates the user's clicking the button with the mouse. This method essentially highlights the button, sends the button's action message to the target object, and then unhighlights the button. If an exception is raised while the target object is processing the action message, the button is unhighlighted before the exception is propagated out of **performClick:**.

See also: **– performKeyEquivalent:**

performKeyEquivalent:

– (BOOL)**performKeyEquivalent:**(NSEvent *)*anEvent*

If the character in *anEvent* matches the button's key equivalent, and the modifier flags in *anEvent* match the key-equivalent modifier mask, **performKeyEquivalent:** simulates the user clicking the button by sending **performClick:** to **self**, and returns YES. Otherwise, **performKeyEquivalent:** does nothing and returns NO. **performKeyEquivalent:** also returns NO in the event that the button is blocked by a modal panel or the button is disabled.

See also: **– keyEquivalentModifierMask**

setAllowsMixedState:

– (void)**setAllowsMixedState**:(BOOL)*flag*

If *flag* is YES, the button has three states: on, off, and mixed. If *flag* is NO, the button has two states: on and off.

See also: **– allowsMixedState**, **– setNextState**

setAlternateImage:

– (void)**setAlternateImage:**(NSImage *)*image*

Sets the image that appears on the button when it's in its alternate state to *image* and, if necessary, redraws the contents of the button. Note that some button types don't display an alternate image.

See also: **– setImage:**, **– setButtonType:**

setAlternateTitle:

– (void)**setAlternateTitle:**(NSString *)*aString*

Sets the string that appears on the button when it's in its alternate state to *aString*. Note that some button types don't display an alternate title.

See also: – **setTitle:**, – **setTitleWithMnemonic:**, – **setButtonType:**, – **setFont:** (NSButtonCell)

setAttributedAlternateTitle:

– (void)**setAttributedAlternateTitle:**(NSAttributedString *)*aString*

Sets the string that appears on the button when it's in its alternate state to the attributed string *aString*. Note that some button types don't display an alternate title.

See also: – **setAttributedTitle:**, – **setButtonType:**, – **setFont:** (NSButtonCell)

setAttributedTitle:

– (void)**setAttributedTitle:**(NSAttributedString *)*aString*

Sets the string that appears on the button when it's in its normal state to the attributed string *aString* and redraws the button. The title is always shown on buttons that don't use their alternate contents when highlighting or displaying their alternate state.

See also: – **setAttributedAlternateTitle:**, – **setButtonType:**, – **setFont:** (NSButtonCell)

setBezelStyle:

– (void)**setBezelStyle:**(NSBezelStyle)*bezelStyle*

Sets the appearance of the border, if the button has one. *bezelStyle* must be one of the following:

Bezel Style	Description
NSNeXTBezelStyle	A rectangular button with a 2 pixel border. It looks like OPENSTEP 4.2 button and is available for backwards compatibility only.
NSPushButtonBezelStyle	A rounded rectangle button, designed for text.
NSSmallIconButtonBezelStyle	A rectangular button with a 2 pixel border, designed for icons.
NSMediumIconButtonBezelStyle	A rectangular button with a 3 pixel border, designed for icons.
NSLargeIconButtonBezelStyle	A rectangular button with a 4 pixel border, designed for icons.

The button uses shading to look like it's sticking out or pushed in. You can set the shading with **setGradientType:**.

If the button is not bordered, the bezel style is ignored.

See also: **– bezelStyle**

setBordered:

– (void)**setBordered:**(BOOL)*flag*

Sets whether the button has a bezeled border. If *flag* is YES, the button displays a border; if NO, the button doesn't display a border. A button's border is not the single line or most other controls' borders; instead, it's a raised bezel. This method redraws the button if **setBordered:** causes the bordered state to change.

setButtonType:

– (void)**setButtonType:**(NSButtonType)*aType*

Sets how the button highlights while pressed and how it shows its state. **setButtonType:** redisplays the button before returning.

The types available are for the most common button types, which are also accessible in Interface Builder. You can configure different behavior with NSButtonCell's **setHighlightsBy:** and **setShowsStateBy:** methods.

aType can be one of eight constants:

Button Type	Description
NSMomentaryLight	While the button is held down it's shown as "lit." This type of button is best for simply triggering actions, as it doesn't show its state; it always displays its normal image or title. This option is called "Momentary Light" in Interface Builder's Button Inspector. This is the default button type.
NSMomentaryPushButton	While the button is held down it's shown as "lit," and also "pushed in" to the screen if the button is bordered. This type of button is best for simply triggering actions, as it doesn't show its state; it always displays its normal image or title. This option is called "Momentary Push" in Interface Builder's Button Inspector.
NSMomentaryChangeButton	While the button is held down, the alternate image and alternate title are displayed. The normal image or title are displayed when the button isn't pressed. This option is called "Momentary Change" in Interface Builder's Button Inspector.
NSPushOnPushOffButton	The first click both highlights and causes the button to be "pushed in" if the button is bordered. A second click returns it to its normal state. This option is called "Push On/Push Off" in Interface Builder's Button Inspector.
NSOnOffButton	The first click highlights the button. A second click returns it to the normal (unhighlighted) state. This option is called "On/Off" in Interface Builder's Button Inspector.
NSToggleButton	The first click highlights the button, while a second click returns it to its normal state. Highlighting is performed by changing to the alternate title or image and showing the button as "pushed in" if the button is bordered. This option is called "Toggle" in Interface Builder's Button Inspector.
NSSwitchButton	This is a variant of NSToggleButton that has no border, with the default image set to "NSSwitch," and the alternate image set to "NSHighlightedSwitch" (these are system bitmaps). This type of button is available as a separate palette item in Interface Builder.
NSRadioButton	Like NSSwitchButton, but the default image is set to "NSRadioButton" and the alternate image is set to "NSHighlightedRadioButton" (these are system bitmaps). This type of button is available as a separate palette item in Interface Builder.

See also: **– setAlternateImage:**, – **setButtonType:** (NSButtonCell), **– setImage:**

setImage:

– (void)**setImage:**(NSImage *)*image*

Sets the button's image to *anImage*, and redraws the button. A button's image is displayed when the button is in its normal state, or all the time for a button that doesn't change its contents when highlighting or displaying its alternate state.

See also: – **setImagePosition:**, – **setAlternateImage:**, – **setButtonType:**

setImagePosition:

– (void)**setImagePosition:**(NSCellImagePosition)*aPosition*

Sets the position of the button's image relative to its title. See the **imagePosition** method description for a listing of possible values for *aPosition*.

setKeyEquivalent:

– (void)**setKeyEquivalent:**(NSString *)*charCode*

Sets the key equivalent character of the button, and redraws the button's interior if it displays a key equivalent instead of an image. The key equivalent isn't displayed if the image position is set to NSNoImage, NSImageOnly or NSImageOverlaps; that is, the button must display both its title and its "image" (the key equivalent in this case), and they must not overlap.

To display a key equivalent on a button, set the image and alternate image to **nil**, then set the key equivalent, then set the image position.

See also: – **performKeyEquivalent:**, – **setAlternateImage:**, – **setImage:**, – **setImagePosition:**, – **setKeyEquivalentFont:** (NSButtonCell)

setKeyEquivalentModifierMask:

– (void)**setKeyEquivalentModifierMask:**(unsigned int)*mask*

Sets the mask indicating the modifier keys to be applied to the button's key equivalent. Mask bits are defined in **NSEvent.h**; only NSControlKeyMask, NSAlternateKeyMask, and NSCommandKeyMask bits are relevant in button key-equivalent modifier masks.

See also: – **setKeyEquivalent:**

setNextState

– (void)**setNextState**

Sets the button to its next state. If the button has three states, it cycles through them in this order: on, off, mixed, on, and so forth. If the button has two states, it toggles between them.

See also: – **allowsMixedState**, – **setAllowsMixedState:**

setPeriodicDelay:interval:

– (void)**setPeriodicDelay:**(float)*delay* **interval:**(float)*interval*

Sets the message delay and interval for the button. These two values are used if the button is configured (by a **setContinuous:** message) to continuously send the action message to the target object while tracking the mouse. *delay* is the amount of time (in seconds) that a continuous button will pause before starting to periodically send action messages to the target object. *interval* is the amount of time (also in seconds) between those messages.

The maximum value allowed for both *delay* and *interval* is 60.0 seconds; if a larger value is supplied, it's ignored and 60.0 seconds is used.

See also: – **setContinuous:** (NSControl)

setState:

– (void)**setState:**(int)*value*

Sets the cell's state to *value*, which can be NSOnState, NSOffState, or NSMixedState. If necessary, this method also redraws the button.

The cell can have two or three states. If it has two, *value* can be NSOffState (the normal or unpressed state) and NSOnState (the alternate or pressed state). If it has three, *value* can be NSOnState (the feature is in effect everywhere), NSOffState (the feature is in effect nowhere), or NSMixedState (the feature is in effect somewhere). Note that if the cell has only two states and *value* is NSMixedState, this method sets the cell's state to NSOnState.

Although using the enumerated constants is preferred, *value* can also be an integer. If the cell has two states, zero is treated as NSOffState, and a non-zero value is treated as NSOnState. If the cell has three states, zero is treated as NSOffState; a negative value, as NSMixedState; and a positive value, as NSOnState.

To check whether the button uses the mixed state, use the method **allowsMixedState**.

setTitle:

– (void)**setTitle:**(NSString *)*aString*

Sets the title displayed by the button when in its normal state to *aString* and, if necessary, redraws the button's contents. This title is always shown on buttons that don't use their alternate contents when highlighting or displaying their alternate state.

See also: – **setAlternateTitle:**, – **setButtonType:**, – **setTitleWithMnemonic:**, – **setFont:** (NSButtonCell)

setTitleWithMnemonic:

– (void)**setTitleWithMnemonic:**(NSString *)*aString*

Sets the title of a button with a character underlined to denote an access key (Windows only). Use an ampersand character to mark the character (the one following the ampersand) to be underlined. For example, the following message causes the "c" in "Receive" to be underlined:

```
[aButton setTitleWithMnemonic:NSLocalizedString(@"Re&ceive")];
```

See also: – **setAlternateTitle:**, – **setButtonType:**, – **setTitle:**, – **setFont:** (NSButtonCell)

setTransparent:

– (void)**setTransparent:**(BOOL)*flag*

Sets whether the button is transparent, and redraws the button if necessary. A transparent button tracks the mouse and sends its action, but doesn't draw. A transparent button is useful for sensitizing an area on the screen so that an action gets sent to a target when the area receives a mouse click.

state

– (int)**state**

Returns the button's state. The button can have two or three states. If it has two, it returns either NSOffState (the normal or unpressed state) or NSOffState (the alternate or pressed state). If it has three, it returns NSOnState (the feature is in effect everywhere), NSOffState (the feature is in effect nowhere), or NSMixedState (the feature is in effect somewhere).

To check whether the button uses the mixed state, use the method **allowsMixedState:**.

title

– (NSString *)**title**

Returns the title displayed on the button when it's in its normal state (this title is always displayed if the button doesn't use its alternate contents for highlighting or displaying the alternate state). Returns the empty string if the button doesn't display a title. By default, a button's title is "Button".

See also: – **alternateTitle**, – **setButtonType:**, – **setTitle:**, – **setTitleWithMnemonic:**

NSButtonCell

Inherits From:	NSActionCell : NSCell : NSObject
Conforms To:	NSCoding (from NSCell) NSCopying (from NSCell) NSObject (from NSObject)
Declared In:	AppKit/NSButtonCell.h

Class Description

NSButtonCell is a subclass of NSActionCell used to implement the user interfaces of push buttons, switches, and radio buttons. It can also be used for any other region of a view that's designed to send a message to a target when clicked. The NSButton subclass of NSControl uses a single NSButtonCell. To create groups of switches or radio buttons, use an NSMatrix holding a set of NSButtonCells.

An NSButtonCell is a two-state cell; it's either "off" or "on," and can be configured to display the two states differently, with a separate title and/or image for either state. The two states are more often referred to as "normal" and "alternate." An NSButtonCell's state is also used as its value, so NSCell methods that set the value (**setIntValue:** and so on) actually set the NSButtonCell's state to "on" if the value provided is non-zero (or non-null for strings), and to "off" if the value is zero or null. Similarly, methods that retrieve the value return 1 for the "on" or alternate state (**stringValue** returns an NSString containing a single character "1"), or 0 for the "off" or normal state (**stringValue** returns an NSString containing a single character "0"). You can also use NSCell's **setState:** and **state** methods to set or retrieve the state directly. After changing the state, send a **display** message to show the NSButtonCell's new appearance. (NSButton does this automatically.)

An NSButtonCell sends its action message to its target once if its view is clicked and it gets the mouse-down event, but can also send the action message continuously as long as the mouse is held down with the cursor inside the NSButtonCell. The NSButtonCell can show that it's being pressed by highlighting in several ways—for example, a bordered NSButtonCell can appear pushed into the screen, or the image or title can change to an alternate form while the NSButtonCell is pressed.

An NSButtonCell can also have a key equivalent (like a menu item). If the NSButtonCell is displayed in the key window, the NSButtonCell gets the first chance to receive events related to key equivalents. This feature is used quite often in modal panels that have an "OK" button. An NSButtonCell can either display a graphical image representing the key equivalent, or you can mark the keyboard "mnemonic" character in the NSButtonCell's title using **setTitleWithMnemonic:**, **setAlternateTitleWithMnemonic:**, or **setAlternateMnemonicLocation:**.

For more information on NSButtonCell's behavior, see the NSButton and NSMatrix class specifications.

Exceptions

In its implementation of the **compare:** method (declared in NSCell), NSButtonCell raises an NSBadComparisonException if the *otherCell* argument is not of the NSButtonCell class.

Method Types

Setting the titles

– alternateMnemonic
– alternateMnemonicLocation
– alternateTitle
– attributedAlternateTitle
– attributedTitle
– setAlternateMnemonicLocation:
– setAlternateTitle:
– setAlternateTitleWithMnemonic:
– setAttributedAlternateTitle:
– setAttributedTitle:
– setFont:
– setTitle:
– setTitleWithMnemonic:
– title

Setting the images

– alternateImage
– imagePosition
– setAlternateImage:
– setImagePosition:

Setting the repeat interval

– getPeriodicDelay:interval:
– setPeriodicDelay:interval:

Setting the key equivalent

– keyEquivalent
– keyEquivalentFont
– keyEquivalentModifierMask
– setKeyEquivalent:
– setKeyEquivalentModifierMask:
– setKeyEquivalentFont:
– setKeyEquivalentFont:size:

Modifying graphic attributes

– bezelStyle
– gradientType
– imageDimsWhenDisabled
– isOpaque
– isTransparent
– setBezelStyle:
– setGradientType:
– setImageDimsWhenDisabled:
– setTransparent:

Displaying

– highlightsBy
– setHighlightsBy:
– setShowsStateBy:
– setButtonType:
– showsStateBy

Simulating a click

– performClick:

Instance Methods

alternateImage

– (NSImage *)**alternateImage**

Returns the image that appears on the button when it's in its alternate state, or **nil** if there is no alternate image. Note that some button types don't display an alternate image. Buttons don't display images by default.

See also: **– image** (NSCell), **– imagePosition**, **– keyEquivalent**, **– setButtonType:**

alternateMnemonic

– (NSString *)**alternateMnemonic**

Returns the character in the alternate title (the title displayed on the button cell when it's in its alternate state) that's marked as the "keyboard mnemonic." If the alternate title doesn't have a keyboard mnemonic, the empty string is returned.

See also: **– alternateMnemonicLocation**, **– mnemonic** (NSCell), **– setAlternateTitleWithMnemonic:**

alternateMnemonicLocation

– (unsigned)**alternateMnemonicLocation**

Returns an unsigned integer indicating the character in the alternate title (the title displayed on the button cell when it's in its alternate state) that's marked as the "keyboard mnemonic." If the alternate title doesn't have a keyboard mnemonic, NSNotFound is returned.

See also: **– alternateMnemonic**, **– mnemonicLocation** (NSCell), **– setAlternateTitleWithMnemonic:**

alternateTitle

– (NSString *)**alternateTitle**

Returns the string that appears on the button when it's in its alternate state, or the empty string if the button doesn't display an alternate title. Note that some button types don't display an alternate title. By default, a button's alternate title is "Button".

See also: **– alternateMnemonic**, **– attributedAlternateTitle**, **– setButtonType:**, **– title**

attributedAlternateTitle

– (NSAttributedString *)**attributedAlternateTitle**

Returns the string that appears on the button when it's in its alternate state as an NSAttributedString, or an empty attributed string if the button doesn't display an alternate title. Note that some button types don't display an alternate title. By default, a button's alternate title is "Button".

See also: **– alternateMnemonic**, **– attributedTitle**, **– setButtonType:**

attributedTitle

– (NSAttributedString *)**attributedTitle**

Returns the string that appears on the button when it's in its normal state as an NSAttributedString, or an empty attributed string if the button doesn't display a title. A button's title is always displayed if the button doesn't use its alternate contents for highlighting or displaying the alternate state. By default, a button's title is "Button".

See also: **– attributedAlternateTitle**, **– mnemonic** (NSCell), **– setButtonType:**

bezelStyle

– (NSBezelStyle)**bezelStyle**

Returns the appearance of the button's border. See **setBezelStyle:** for the list of the possible values.

See also: **– setBezelStyle:**

getPeriodicDelay:interval:

– (void)**getPeriodicDelay:**(float *)*delay* **interval:**(float *)*interval*

Returns by reference the delay and interval periods for a continuous button. *delay* is the amount of time (in seconds) that the button will pause before starting to periodically send action messages to the target object. *interval* is the amount of time (also in seconds) between those messages.

Default delay and interval values are taken from a user's defaults (60 seconds maximum for each); if the user hasn't specified default values, *delay* defaults to 0.4 seconds and *interval* defaults to 0.075 seconds.

See also: – **isContinuous** (NSCell)

gradientType

– (NSGradientType)**gradientType**

Returns gradient of the button's border. See **setGradientType:** for the list of the possible values.

highlightsBy

– (int)**highlightsBy**

Returns the logical OR of flags that indicate the way the button cell highlights when it receives a mouse-down event. See **setHighlightsBy:** for the list of flags.

See also: – **showsStateBy**

imageDimsWhenDisabled

– (BOOL)**imageDimsWhenDisabled**

Returns whether the button cell's image and text appear "dim" when the button cell is disabled. By default, all button types except NSSwitchButton and NSRadioButton do dim when disabled. When NSSwitchButtons and NSRadioButtons are disabled, only the associated text dims.

See also: **– setButtonType:**

imagePosition

– (NSCellImagePosition)**imagePosition**

Returns the position of the button's image relative to its title. The return value is one of the following (these are defined in **NSCell.h**):

Return Value	Meaning
NSNoImage	The button doesn't display an image (this is the default)
NSImageOnly	The button displays an image, but not a title
NSImageLeft	The image is to the left of the title
NSImageRight	The image is to the right of the title
NSImageBelow	The image is below the title
NSImageAbove	The image is above the title
NSImageOverlaps	The image overlaps the title

If the title is above, below, or overlapping the image, or if there is no image, the text is horizontally centered within the button.

See also: – **setButtonType:**, **– setImage:** (NSCell), **– setTitle:**

isOpaque

– (BOOL)**isOpaque**

Returns YES if the button cell draws over every pixel in its frame, NO if not. The button cell is opaque only if it isn't transparent and if it has a border.

See also: **– isTransparent**

isTransparent

– (BOOL)**isTransparent**

Returns YES if the button is transparent, NO otherwise. A transparent button never draws itself, but it receives mouse-down events and tracks the mouse properly.

See also: **– isOpaque**

keyEquivalent

– (NSString *)**keyEquivalent**

Returns the key-equivalent character of the button, or the empty string if one hasn't been defined. Buttons don't have a default key equivalent.

See also: **– keyEquivalentFont**

keyEquivalentFont

– (NSFont *)**keyEquivalentFont**

Returns the font used to draw the key equivalent, or **nil** if the button cell doesn't have a key equivalent. The default font is the same as that used to draw the title.

See also: **– setFont:**

keyEquivalentModifierMask

– (unsigned int)**keyEquivalentModifierMask**

Returns the mask indicating the modifier keys that are applied to the button's key equivalent. Mask bits are defined in **NSEvent.h**; only NSControlKeyMask, NSAlternateKeyMask, and NSCommandKeyMask bits are relevant in button key-equivalent modifier masks.

See also: **– keyEquivalent**

performClick:

– (void)**performClick:**(id)*sender*

Simulates the user's clicking the button with the mouse. This method essentially highlights the button, sends the button's action message to the target object, and then unhighlights the button. If an exception is raised while the target object is processing the action message, the button is unhighlighted before the exception is propagated out of **performClick:**.

setAlternateImage:

– (void)**setAlternateImage:**(NSImage *)*image*

Sets the image that appears on the button when it's in its alternate state to *image* and, if necessary, redraws the contents of the button. Note that some button types don't display an alternate image.

See also: – **setImage:** (NSCell), – **setButtonType:**

setAlternateMnemonicLocation:

– (void)**setAlternateMnemonicLocation:**(unsigned)*location*

Sets the character in the alternate title (the title displayed on the button cell when it's in its alternate state) that's to be marked as the "keyboard mnemonic." The character specified by *location* will be underlined; *location* can be any integer from 0 to 254. If you don't want the alternate title to have a keyboard mnemonic, specify a location of NSNotFound.

setAlternateMnemonicLocation: doesn't cause the button cell to be redisplayed.

See also: **– setAlternateTitleWithMnemonic:**

setAlternateTitle:

– (void)**setAlternateTitle:**(NSString *)*aString*

Sets the title that's displayed on the button when it's in its alternate state to *aString*. Note that some button types don't display an alternate title.

See also: **– setAlternateMnemonicLocation:**, **– setAlternateTitleWithMnemonic:**, **– setTitle:**, **– setButtonType:**, **– setFont:**

setAlternateTitleWithMnemonic:

– (void)**setAlternateTitleWithMnemonic:**(NSString *)*aString*

Sets the title that is displayed on the button cell when it's in its alternate state to *aString*, taking into account the fact that an embedded "&" character is not a literal but instead marks the alternate state's "keyboard mnemonic." The character in the title that immediately follows the "&" character will be underlined.

If necessary, **setAlternateTitleWithMnemonic:** redraws the button cell. Note that some button types don't display an alternate title.

See also: **– setAlternateMnemonicLocation:**, **– setTitleWithMnemonic:**

setAttributedAlternateTitle:

– (void)**setAttributedAlternateTitle:**(NSAttributedString *)*aString*

Sets the string that appears on the button when it's in its alternate state to the attributed string *aString*. Note that some button types don't display an alternate title.

See also: **– setAlternateMnemonicLocation:**, **– setAlternateTitleWithMnemonic:**, **– setAttributedTitle:**, **– setButtonType:**, **– setFont:**

setAttributedTitle:

– (void)**setAttributedTitle:**(NSAttributedString *)*aString*

Sets the string that appears on the button when it's in its normal state to the attributed string *aString* and redraws the button. The title is always shown on buttons that don't use their alternate contents when highlighting or displaying their alternate state.

See also: – **setAttributedAlternateTitle:**, – **setButtonType:**, – **setFont:**, **– setMnemonicLocation:** (NSCell)

setBezelStyle:

– (void)**setBezelStyle:**(NSBezelStyle)*bezelStyle*

Sets the appearance of the border, if the button has one. *bezelStyle* must be one of the following:

Bezel Style	Description
NSNeXTBezelStyle	A rectangular button with a 2 pixel border. It looks like OPENSTEP 4.2 button and is available for backwards compatibility only.
NSPushButtonBezelStyle	A rounded rectangle button, designed for text.
NSSmallIconButtonBezelStyle	A rectangular button with a 2 pixel border, designed for icons.
NSMediumIconButtonBezelStyle	A rectangular button with a 3 pixel border, designed for icons.
NSLargeIconButtonBezelStyle	A rectangular button with a 4 pixel border, designed for icons.

The button uses shading to look like it's sticking out or pushed in. You can set the shading with **setGradientType:**.

If the button is not bordered, the bezel style is ignored.

See also: **– bezelStyle**

setButtonType:

– (void)**setButtonType:**(NSButtonType)*aType*

Sets how the button highlights while pressed and how it shows its state. **setButtonType:** redisplays the button before returning.

The types available are for the most common button types, which are also accessible in Interface Builder; you can configure different behavior with the **setHighlightsBy:** and **setShowsStateBy:** methods.

aType can be one of eight constants:

Button Type	Description
NSMomentaryLight	While the button is held down it's shown as "lit." This type of button is best for simply triggering actions, as it doesn't show its state; it always displays its normal image or title. This option is called "Momentary Light" in Interface Builder's Button Inspector. This is the default button type.
NSMomentaryPushButton	While the button is held down it's shown as "lit," and also "pushed in" to the screen if the button is bordered. This type of button is best for simply triggering actions, as it doesn't show its state; it always displays its normal image or title. This option is called "Momentary Push" in Interface Builder's Button Inspector.
NSMomentaryChangeButton	While the button is held down, the alternate image and alternate title are displayed. The normal image and title are displayed when the button isn't pressed. This option is called "Momentary Change" in Interface Builder's Button Inspector.
NSPushOnPushOffButton	The first click both highlights and causes the button to be "pushed in" if the button is bordered. A second click returns it to its normal state. This option is called "Push On/Push Off" in Interface Builder's Button Inspector.
NSOnOffButton	The first click highlights the button. A second click returns it to the normal (unhighlighted) state. This option is called "On/Off" in Interface Builder's Button Inspector.
NSToggleButton	The first click highlights the button, while a second click returns it to its normal state. Highlighting is performed by changing to the alternate title or image and showing the button as "pushed in" if the button is bordered. This option is called "Toggle" in Interface Builder's Button Inspector.
NSSwitchButton	This is a variant of NSToggleButton that has no border, with the default image set to "NSSwitch," and the alternate image set to "NSHighlightedSwitch" (these are system bitmaps). This type of button is available as a separate palette item in Interface Builder.
NSRadioButton	Like NSSwitchButton, but the default image is set to "NSRadioButton" and the alternate image is set to "NSHighlightedRadioButton" (these are system bitmaps). This type of button is available as a separate palette item in Interface Builder.

See also: **– setAlternateImage:**, – **setButtonType:**, **– setImage:** (NSCell)

setFont:

– (void)**setFont:**(NSFont *)*fontObj*

Sets the font used to display the title and alternate title. Does nothing if the button cell has no title or alternate title.

If the button cell has a key equivalent, its font is not changed, but the key equivalent's font size is changed to match the new title font.

See also: **– font** (NSCell), **– setKeyEquivalentFont:**, **– setKeyEquivalentFont:size:**

setGradientType:

– (void)**setGradientType:**(NSGradientType)*gradientType*

Sets the type of gradient to use for the button. If the button has no border, this method has no affect on its appearance.

gradientType can be one of the following contants:

Value	Description
NSGradientNone	There is no gradient, so the button looks flat.
NSGradientConcanveWeak	The top left corner is light gray and the bottom right corner is dark gray, so the button appears to be pushed in.
NSGradientConcaveStrong	As with NSGradientConcanveWeak, the top left corner is light gray and the bottom right corner is dark gray, but the difference between the grays is greater, so the appearance of being pushed-in is stronger.
NSGradientConvexWeak	The top left corner is dark gray and the bottom right corner is light gray, so the button appears to be sticking out.
NSGradientConcaveStrong	As with NSGradientConvexWeak, the top left corner is dark gray and the bottom right corner is light gray, but the difference between the grays is greater, so the appearance of sticking out is stronger.

See also: **– gradientType**

setHighlightsBy:

– (void)**setHighlightsBy:**(int)*aType*

Sets the way the button cell highlights itself while pressed. *aType* can be the logical OR of one or more of the following constants:

Value	Description
NSNoCellMask	The button cell doesn't change. This flag is ignored if any others are set in *aType*.
NSPushInCellMask	The button cell "pushes in" when pressed if it has a border. This is the default behavior.
NSContentsCellMask	The button cell displays its alternate icon and/or title.
NSChangeGrayCellMask	The button cell swaps the "control color" (NSColor's controlColor) and white pixels on the its background and icon.
NSChangeBackgroundCellMask	Same as NSChangeGrayCellMask, but only background pixels are changed.

If both NSChangeGrayCellMask and NSChangeBackgroundCellMask are specified, both are recorded, but which behavior is used depends on the button cell's image. If the button has no image, or if the image has no alpha (transparency) data, NSChangeGrayCellMask is used. If the image does have alpha data, NSChangeBackgroundCellMask is used; this allows the color swap of the background to show through the image's transparent pixels.

See also: – **setShowsStateBy:**

setImageDimsWhenDisabled:

– (void)**setImageDimsWhenDisabled:**(BOOL)flag

Sets whether the button cell's image and text appear "dim" when the button cell is disabled. By default, all button types except NSSwitchButton and NSRadioButton do dim when disabled. When NSSwitchButtons and NSRadioButtons are disabled, only the associated text associated dims. The default setting for this condition is reasserted whenever you invoke **setButtonType:**, so be sure to specify the button cell's type before you invoke **setImageDimsWhenDisabled:**.

setImagePosition:

– (void)**setImagePosition:**(NSCellImagePosition)*aPosition*

Sets the position of the button's image relative to its title. See the **imagePosition** method description for a listing of possible values for *aPosition*.

setKeyEquivalent:

– (void)**setKeyEquivalent:**(NSString *)*aKeyEquivalent*

Sets the key equivalent character of the button, and redraws the button's inside if it displays a key equivalent instead of an image. The key equivalent isn't displayed if the image position is set to NSNoImage, NSImageOnly or NSImageOverlaps; that is, the button must display both its title and its "image" (the key equivalent in this case), and they must not overlap.

To display a key equivalent on a button, set the image and alternate image to **nil**, then set the key equivalent, then set the image position.

See also: – **setAlternateImage:**, – **setImage:** (NSCell), – **setImagePosition:**, **– setKeyEquivalentFont:**

setKeyEquivalentFont:

– (void)**setKeyEquivalentFont:**(NSFont *)*fontObj*

Sets the font used to draw the key equivalent, and redisplays the button cell if necessary. Does nothing if the button cell doesn't have a key equivalent associated with it. The default font is the same as that used to draw the title.

See also: **– setFont:**

setKeyEquivalentFont:size:

– (void)**setKeyEquivalentFont:**(NSString *)*fontName* **size:**(float)*fontSize*

Sets by name and size the font used to draw the key equivalent, and redisplays the button cell if necessary. Does nothing if the button cell doesn't have a key equivalent associated with it. The default font is the same as that used to draw the title.

See also: **– setFont:**

setKeyEquivalentModifierMask:

– (void)**setKeyEquivalentModifierMask:**(unsigned int)*mask*

Sets the mask indicating the modifier keys to be applied to the button's key equivalent. Mask bits are defined in **NSEvent.h**; only NSControlKeyMask, NSAlternateKeyMask, and NSCommandKeyMask bits are relevant in button key-equivalent modifier masks.

See also: **– setKeyEquivalent:**

setPeriodicDelay:interval:

– (void)**setPeriodicDelay:**(float)*delay* **interval:**(float)*interval*

Sets the message delay and interval for the button. These two values are used if the button is configured (by a **setContinuous:** message) to continuously send the action message to the target object while tracking the mouse. *delay* is the amount of time (in seconds) that a continuous button will pause before starting to periodically send action messages to the target object. *interval* is the amount of time (also in seconds) between those messages.

The maximum value allowed for both *delay* and *interval* is 60.0 seconds; if a larger value is supplied, it's ignored and 60.0 seconds is used.

See also: – **setContinuous:** (NSCell)

setShowsStateBy:

– (void)**setShowsStateBy:**(int)*aType*

Sets the way the button cell indicates its alternate state. *aType* should be the logical OR of one or more of the following constants:

Value	Description
NSNoCellMask	The button cell doesn't change. This mask is ignored if any others are set in *aType*. This is the default.
NSContentsCellMask	The button cell displays its alternate icon and/or title.
NSChangeGrayCellMask	The button cell swaps the "control color" (NSColor's controlColor) and white pixels on its background and icon.
NSChangeBackgroundCellMask	Same as NSChangeGrayCellMask, but only the background pixels are changed.

If both NSChangeGrayCellMask and NSChangeBackgroundCellMask are specified, both are recorded, but the actual behavior depends on the button cell's image. If the button has no image, or if the image has no alpha (transparency) data, NSChangeGrayCellMask is used. If the image exists and has alpha data, NSChangeBackgroundCellMask is used; this allows the color swap of the background to show through the image's transparent pixels.

See also: – **setHighlightsBy:**, **– showsStateBy**

setTitle:

– (void)**setTitle:**(NSString *)*aString*

Sets the title displayed by the button cell when in its normal state to *aString* and, if necessary, redraws the button's contents. This title is always shown on buttons that don't use their alternate contents when highlighting or displaying their alternate state.

See also: – **setAlternateTitle:**, – **setButtonType:**, – **setFont:**, **– setTitleWithMnemonic:**

setTitleWithMnemonic:

– (void)**setTitleWithMnemonic:**(NSString *)*aString*

Sets the title displayed on the button cell when it's in its normal state to *aString*, taking into account the fact that an embedded "&" character is not a literal but instead marks the normal state's "keyboard mnemonic." If necessary, **setTitleWithMnemonic:** redraws the button cell. This title is always shown on buttons that don't use their alternate contents when highlighting or displaying their alternate state.

See also: **– setAlternateTitleWithMnemonic:**, **– setMnemonicLocation:** (NSCell),
– setTitleWithMnemonic: (NSCell)

setTransparent:

– (void)**setTransparent:**(BOOL)*flag*

Sets whether the button is transparent, and redraws the button if necessary. A transparent button tracks the mouse and sends its action, but doesn't draw. A transparent button is useful for sensitizing an area on the screen so that an action gets sent to a target when the area receives a mouse click.

showsStateBy

– (int)**showsStateBy**

Returns the logical OR of flags that indicate the way the button cell shows its alternate state. See **setShowsStateBy:** for the list of flags.

See also: – **highlightsBy**, **– setShowsStateBy:**

title

– (NSString *)**title**

Returns the title displayed on the button when it's in its normal state (this title is always displayed if the button doesn't use its alternate contents for highlighting or displaying the alternate state). Returns the empty string if the button doesn't display a title. By default, a button's title is "Button".

See also: – **alternateTitle**, – **setButtonType:**, **– mnemonic** (NSCell), **– mnemonicLocation** (NSCell)

NSCachedImageRep

Inherits From: NSImageRep : NSObject

Conforms To: NSCoding (from NSImageRep)
NSCopying (from NSImageRep)
NSObject (from NSObject)

Declared In: AppKit/NSImageRep.h

Class Description

NSCachedImageRep, a subclass of NSImageRep, defines an object that stores its source data as a rendered image in a window, typically a window that stays off-screen. The only data that's available for reproducing the image is the image itself. Thus an NSCachedImageRep differs from the other kinds of NSImageReps defined in the Application Kit, all of which can reproduce an image from the information originally used to draw it. Instances of this class are generally used indirectly, through an NSImage object.

See "Caching Representations" in the NSImage class description for more information.

Method Types

Initializing an NSCachedImageRep

– initWithSize:depth:separate:alpha:
– initWithWindow:rect:

Getting the representation

– rect
– window

Instance Methods

initWithSize:depth:separate:alpha:

– (id)**initWithSize:**(NSSize)*size*
depth:(NSWindowDepth)*depth*
separate:(BOOL)*flag*
alpha:(BOOL)*alpha*

Initializes a new NSCachedImageRep for an image of the specified size and depth. *flag* indicates whether the image will get its own unique cache, instead of possibly sharing one with other images. For best performance (although it's not essential), *alpha* should be set according to whether the image will have a channel for transparency information.

See also: **– setAlpha:** (NSImageRep), **– setBitsPerSample:** (NSImageRep), **– setCacheDepthMatchesImageDepth:** (NSImage), **– setCachedSeparately:** (NSImage)

initWithWindow:rect:

– (id)**initWithWindow:**(NSWindow *)*aWindow* **rect:**(NSRect)*aRect*

Initializes the receiver, a new NSCachedImageRep instance, for an image that will be rendered within the *aRect* rectangle in the window *aWindow*, and returns the initialized object. The rectangle is specified in *aWindow*'s base coordinate system. The size of the image is set from the size of the rectangle. The window is retained.

You must draw the image in the rectangle yourself; there are no NSCachedImageRep methods for this purpose.

See also: **– size** (NSImageRep)

rect

– (NSRect)**rect**

Returns the rectangle where the image is cached.

See also: **– size** (NSImageRep)

window

– (NSWindow *)**window**

Returns the window where the image is cached.

NSCell

Inherits From:	NSObject
Conforms To:	NSCoding (NSObject), NSCopying (NSObject), NSObject (NSObject)
Declared In:	AppKit/NSCell.h

Class Description

The NSCell class provides a mechanism for displaying text or images in an NSView without the overhead of a full NSView subclass. In particular, it provides much of the functionality of the NSText class by providing access to a shared NSText object used by all instances of NSCell in an application. NSCells are also extremely useful for placing text or images at various locations in a custom subclass of NSView.

NSCell is used heavily by most of the NSControl classes to implement their internal workings. For example, NSSlider uses an NSSliderCell, NSTextField uses an NSTextFieldCell, and NSBrowser uses an NSBrowserCell. Sending a message to the NSControl is often simpler than dealing directly with the corresponding NSCell. For instance, NSControls typically invoke **updateCell:** (causing the cell to be displayed) after changing a cell attribute; whereas if you directly call the corresponding method of the NSCell, the NSCell might not automatically display itself again.

Some subclasses of NSControl (notably NSMatrix) group NSCells in an arrangement where they act together in some cooperative manner. Thus, with an NSMatrix, you can implement a uniformly sized group of radio buttons without needing an NSView for each button (and without needing an NSText object as the field editor for the text on each button).

The NSCell class provides primitives for displaying text or an image, editing text, setting and getting object values, maintaining state, highlighting, and tracking the mouse. NSCell's method **trackMouse:inRect: ofView:untilMouseUp:** implements the mechanism that sends action messages to target objects. However, NSCell implements target/action features abstractly, deferring the details of implementation to NSActionCell and its subclasses.

Object Values and Formatters

Every NSCell that displays text has a value associated with it. The NSCell stores that value as an object of potentially any type, displays it as an NSString, and returns it as a primary value or string object, according to what's requested (**intValue**, **floatValue**, **stringValue**, and so on). Formatters are objects associated with NSCells (through **setFormatter:**) that translate a cell's object value to its it textual representation and that convert what users type into the underlying object. NSCells have built-in formatters to handle common

string and numeric (**int**, **float**, **double**) translations. In addition, you can specify date and numeric types more precisely with **setEntryType:** and specify floating-point format characteristics with **setFloatingPointFormat:left:right:**. You can also implement your own formatters to provide specialized object translation; see the NSFormatter specification for more information.

The text that an NSCell displays and stores can be an attributed string. Several methods help to set and get attributed-string values, including s**etAttributedStringValue:** and **setImportsGraphics:**.

Cell States

For some subclasses of NSCell, such as an NSButtonCell, the object's value is its state. It can have either two states—NSOnState and NSOffState—or three states—NSOnState, NSOffState, and NSMixedState. A mixed state is useful for a checkbox or radio button that reflects the status of a feature. For example, suppose you have a checkbox that makes the selected text bold. If all the selected text is bold, it's on. If none of the selected text is bold, it's off. If the text has a combination of bold and plain text, it's mixed. Now suppose you click the checkbox. If you turn it on, all the text becomes bold. If you turn it off, all the text becomes plain. If you select the mixed state, the text remains as it is.

By default, an NSCell has two states. You can allow the third state with the method **setAllowsMixedState**. To set the button's state directly, use **setState:**. To cycle through all available states, use **setNextState**.

Represented Objects

Represented objects are objects that an NSCell stands for. (They're not to be confused with an NSCell's object value, which *is* the value of the cell.) By setting a represented object for an NSCell (using **setRepresentedObject:**) you make an association between the NSCell and that object. For instance, you could have a pop-up list, each cell of which lists a color as its title; when the user selects a cell, the represented NSColor object is displayed in a color well. This feature is solely for the developer's convenience. The cell itself does not use the represented object, except to archive and restore it.

Subclassing NSCell

The **initImageCell:** method is the designated initializer for NSCells that display images. The **initTextCell:** method is the designated initializer for NSCells that display text. Override one or both of these methods if you implement a subclass of NSCell that performs its own initialization. If you need to use target and action behavior, you may prefer to subclass NSActionCell or one of its subclasses, which provide the default implementation of this behavior.

If you want to implement your own mouse-tracking or mouse-up behavior, consider overriding **startTrackingAt:inView:**, **continueTracking:at:inView:**, and **stopTracking:at:inView:mouseIsUp:**. If you want to implement your own drawing, override **drawWithFrame:inView:** or **drawInteriorWithFrame:inView:**.

If the subclass contains instance variables that hold pointers to objects, consider overriding **copyWithZone:** to duplicate the objects. The default version copies only pointers to the objects.

For more information on how NSCell is used, see the NSControl class specification.

Method Types

Initializing an NSCell

– initImageCell:
– initTextCell:

Setting and getting cell values

– setObjectValue:
– objectValue
– hasValidObjectValue
– setIntValue:
– intValue
– setStringValue:
– stringValue
– setDoubleValue:
– doubleValue
– setFloatValue:
– floatValue

Setting and getting cell attributes

– setCellAttribute:to:
– cellAttribute:
– setType:
– type
– setEnabled:
– isEnabled
– setBezeled:
– isBezeled
– setBordered:
– isBordered
– isOpaque

Setting the state

– allowsMixedState
– nextState
– setAllowsMixedState:
– setNextState
– setState:
– state

Modifying textual attributes of cells

– setEditable:
– isEditable
– setSelectable:
– isSelectable
– setScrollable:
– isScrollable
– setAlignment:
– alignment
– setFont:
– font
– setWraps:
– wraps
– setAttributedStringValue:
– attributedStringValue
– setAllowsEditingTextAttributes:
– allowsEditingTextAttributes
– setImportsGraphics:
– importsGraphics
– setUpFieldEditorAttributes:

Setting the target and action

– setAction:
– action
– setTarget:
– target
– setContinuous:
– isContinuous
– sendActionOn:

Setting and getting an image

– setImage:
– image

Assigning a tag

– setTag:
– tag

Formatting and validating data

– setFormatter:
– formatter
– setEntryType:
– entryType
– isEntryAcceptable:
– setFloatingPointFormat:left:right:

Managing menus for cells

+ defaultMenu
– setMenu:
– menu
– menuForEvent:inRect:ofView:

Comparing cells

– compare:

Making cells respond to keyboard events

– acceptsFirstResponder
– setShowsFirstResponder:
– showsFirstResponder
– setTitleWithMnemonic:
– mnemonic
– refusesFirstResponder
– setMnemonicLocation:
– setRefusesFirstResponder:
– mnemonicLocation
– performClick:

Deriving values from other cells

– takeObjectValueFrom:
– takeIntValueFrom:
– takeStringValueFrom:
– takeDoubleValueFrom:
– takeFloatValueFrom:

Representing an object with a cell

– setRepresentedObject:
– representedObject

Tracking the mouse

– trackMouse:inRect:ofView:untilMouseUp:
– startTrackingAt:inView:
– continueTracking:at:inView:
– stopTracking:at:inView:mouseIsUp:
– mouseDownFlags
+ prefersTrackingUntilMouseUp
– getPeriodicDelay:interval:

Managing the cursor

– resetCursorRect:inView:

Handling keyboard alternatives

– keyEquivalent

Determining cell sizes

– calcDrawInfo:
– cellSize
– cellSizeForBounds:
– drawingRectForBounds:
– imageRectForBounds:
– titleRectForBounds:

Drawing and highlighting cells

– drawWithFrame:inView:
– drawInteriorWithFrame:inView:
– controlView
– highlight:withFrame:inView:
– isHighlighted

Editing and selecting cell text

– editWithFrame:inView:editor:delegate:event:
– selectWithFrame:inView:editor:delegate:start:length:
– sendsActionOnEndEditing
– setSendsActionOnEndEditing:
– endEditing:

Class Methods

defaultMenu

+ (NSMenu *)**defaultMenu**

Returns the default menu for instances of the receiver. The default implementation returns **nil**.

See also: – **menu**, – **setMenu:**

prefersTrackingUntilMouseUp

+ (BOOL)**prefersTrackingUntilMouseUp**

The default implementation returns NO, so tracking stops when the mouse leaves the NSCell; subclasses may override.

See also: – **trackMouse:inRect:ofView:untilMouseUp:**

Instance Methods

acceptsFirstResponder

– (BOOL)**acceptsFirstResponder**

The default implementation returns YES if the cell is enabled and **refusesFirstResponder** returns NO; subclasses can override.

See also: – **performClick:**, – **setShowsFirstResponder:**, – **setTitleWithMnemonic:**

action

– (SEL)**action**

Implemented by NSActionCell and its subclasses to return the selector of the cell's action method. The default implementation returns a null selector.

See also: – **setAction:**, – **setTarget:**, – **target**

alignment

– (NSTextAlignment)**alignment**

Returns the alignment of text in the cell: NSLeftTextAlignment, NSRightTextAlignment, NSCenterTextAlignment, NSJustifiedTextAlignment, or NSNaturalTextAlignment.

See also: – **setAlignment:**

allowsEditingTextAttributes

– (BOOL)**allowsEditingTextAttributes**

Returns whether the receiver allows the editing of textual attributes.

See also: – **setAllowsEditingTextAttributes:**

allowsMixedState

– (BOOL)**allowsMixedState**

Returns YES if the button has three states: on, off, and mixed. Returns NO if the button has two states: on and off.

See also: – **nextState**, – **setAllowsMixedState:**, – **setNextState**

attributedStringValue

– (NSAttributedString *)**attributedStringValue**

Returns the value of the receiver as an attributed string, using the cell's formatter object (if one exists) to create the attributed string. The textual attributes are determined by the default paragraph style, the receiver's font and alignment, and whether the receiver is enabled and scrollable.

See also: – **setAttributedStringValue:**

calcDrawInfo:

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

Implemented by subclasses to recalculate drawing sizes with reference to *aRect*. Objects (such as NSConrols) that manage NSCells generally maintain a flag that informs them if any of their cells has been modified in such a way that the location or size of the cell should be recomputed. If so, NSControl's **calcSize** method is automatically invoked prior to the display of the NSCell, and that method invokes the NSCell's **calcDrawInfo:** method. The default implementation does nothing.

See also: – **cellSize**, – **drawingRectForBounds:**

cellAttribute:

– (int)**cellAttribute:**(NSCellAttribute)*aParameter*

Depending on *aParameter*, returns a setting for a cell attribute, such as the receiver's state, and whether it's disabled, editable, or highlighted.

See also: – **setCellAttribute:to:**

cellSize

– (NSSize)**cellSize**

Returns the minimum size needed to display the NSCell, taking account of the size of the image or text within a certain offset determined by border type. If the receiving cell is neither of image or text type, an extremely large size is returned; if the receiving cell is of image type, and no image has been set, an extremely small size is returned.

See also: – **drawingRectForBounds:**

cellSizeForBounds:

– (NSSize)**cellSizeForBounds:**(NSRect)*aRect*

Returns the minimum size needed to display the NSCell, taking account of the size of the image or text within an offset determined by border type. If the receiving cell is of text type, the text is resized to fit within *aRect* (as much as *aRect* is within the bounds of the cell). If the receiving cell is neither of image or text type, an extremely large size is returned; if the receiving cell is of image type, and no image has been set, an extremely small size is returned.

See also: – **drawingRectForBounds:**

compare:

– (NSComparisonResult)**compare:**(id)*otherCell*

Compares the string values of this cell and *otherCell* (which must be a kind of NSCell), disregarding case. Raises NSBadComparisonException if *otherCell* is not of the NSCell class or if one of the cells being compared is not a text-type cell.

continueTracking:at:inView:

– (BOOL)**continueTracking:**(NSPoint)*lastPoint*
at:(NSPoint)*currentPoint*
inView:(NSView *)*controlView*

Returns whether mouse-tracking should continue in the receiving cell based on *lastPoint* and *currentPoint* within *controlView* (*currentPoint* is the current location of the mouse while *lastPoint* is either the initial location of the mouse or the previous *currentPoint*). This method is invoked in **trackMouse:inRect: ofView:untilMouseIsUp:**. The default implementation returns YES if the cell is set to continuously send action messages to its target when the mouse is down or is being dragged. Subclasses can override this method to provide more sophisticated tracking behavior.

See also: – **startTrackingAt:inView:**, – **stopTracking:at:inView:mouseIsUp:**

controlView

– (NSView *)**controlView**

Implemented by subclasses to return the NSView last drawn in (normally an NSControl). The default implementation returns **nil**.

See also: – **drawWithFrame:inView:**

doubleValue

– (double)**doubleValue**

Returns the NSCell's value as a **double**. If the receiver is not a text-type cell or the cell value is not scannable, the method returns zero.

drawInteriorWithFrame:inView:

– (void)**drawInteriorWithFrame:**(NSRect)*cellFrame* **inView:**(NSView *)*controlView*

Draws the "inside" of the receiving cell; this includes the image or text within the NSCell's frame in *controlView* (usually the cell's NSControl) but excludes the border. *cellFrame* is the frame of the NSCell or (in some cases) a portion of it. Text-type NSCells display their contents in a rectangle slightly inset from *cellFrame* using a global NSText object; image-type NSCells display their contents centered within *cellFrame*. If the proper attributes are set, it also displays the dotted-line rectangle to indicate first responder and highlights the cell. This method is invoked from NSControl's **drawCellInside:** to visually update the what the NSCell displays when its contents change. This drawing is minimal, and becomes more complex in objects such as NSButtonCell and NSSliderCell.

Subclasses often override this method to provide more sophisticated drawing of cell contents. Because **drawWithFrame:inView:** invokes **drawInteriorWithFrame:inView**: after it draws the NSCell's border, don't invoke **drawWithFrame:inView:** in your override implementation.

See also: – **isHighlighted**, – **setShowsFirstResponder:**

drawWithFrame:inView:

– (void)**drawWithFrame:**(NSRect)*cellFrame* **inView:**(NSView *)*controlView*

Draws the receiver's regular or bezeled border (if those attributes are set) and then draws the interior of the cell by invoking **drawInteriorWithFrame:inView:**.

drawingRectForBounds:

– (NSRect)**drawingRectForBounds:**(NSRect)*theRect*

Returns the rectangle within which the cell draws itself; this rectangle is slightly inset from *aRect* on all sides to take the border into account.

See also: – **calcSize** (NSControl)

editWithFrame:inView:editor:delegate:event:

– (void)**editWithFrame:**(NSRect)*aRect*
inView:(NSView *)*controlView*
editor:(NSText *)*textObj*
delegate:(id)*anObject*
event:(NSEvent *)*theEvent*

Begins editing of the receiver's text by using the field editor textObj; usually invoked in response to a mouse-down event. *aRect* must be the rectangle used for displaying the NSCell. *theEvent* is the NSMouseDown event. *anObject* is made the delegate of *textObj*, and so will receive various NSText delegation and notification messages.

If the receiver isn't a text-type NSCell, no editing is performed. Otherwise, *textObj* is sized to *aRect* and its superview is set to *aView*, so that it exactly covers the NSCell. Then it's activated and editing begins. It's the responsibility of the delegate to end the editing when responding to **textShouldEndEditing:**; in doing this, it should remove any data from *textObj*.

See also: – **endEditing:**, – **selectWithFrame:inView:editor:delegate:start:length:**

endEditing:

– (void)**endEditing:**(NSText *)*textObj*

Ends any editing of text occurring in the receiver begun with **editWithFrame:inView:editor:delegate:event:** and **selectWithFrame:inView:editor:delegate:start:length:**.

entryType

– (int)**entryType**

Returns the type of data the user can type into the receiver. If the receiver is not a text-type cell, or if no type has been set, NSAnyType is returned. See **setEntryType:** for a list of type constants.

See also: – **isEntryAcceptable:**

floatValue

– (float)**floatValue**

Returns the NSCell's value as a **float**. If the receiver is not a text-type cell or the cell value is not scannable, the method returns zero.

font

– (NSFont *)**font**

Returns the font used to display text in the receiving cell or **nil** if the receiver is not a text-type cell.

See also: – **setFont:**

formatter

– (id)**formatter**

Returns the formatter object (a kind of NSFormatter) associated with the cell. This object handles translation of the cell's contents between it's on-screen representation and its object value.

See also: – **setFormatter:**

getPeriodicDelay:interval:

– (void)**getPeriodicDelay:**(float *)*delay* **interval:**(float *)*interval*

Returns initial delay and repeat values for continuous sending of action messages to target objects. Subclasses can override to supply their own delay and interval values.

See also: – **isContinuous**, – **setContinuous:**

hasValidObjectValue

– (BOOL)**hasValidObjectValue**

Returns whether the object associated with the receiver has a valid object value. A valid object value is one that the receiver's formatter can "understand." Objects that are "invalid" have been rejected by the formatter, but accepted by the delegate of the receiver's NSControl (in **control:didFailToFormatString: errorDescription:**).

See also: – **objectValue**, – **setObjectValue:**

highlight:withFrame:inView:

– (void)**highlight:**(BOOL)*flag*
withFrame:(NSRect)*cellFrame*
inView:(NSView *)*controlView*

If the receiver's highlight status is different from *flag*, sets that status to *flag* and, if *flag* is YES, highlights the rectangle *cellFrame* in the NSControl (*controlView*).

Note that NSCell's highlighting does not appear when highlighted cells are printed (although instances of NSTextFieldCell, NSButtonCell, and others can print themselves highlighted). Generally, you cannot depend on highlighting being printed because implementations of this method may choose (or not choose) to use transparency.

See also: – **drawWithFrame:inView:**, – **isHighlighted**

image

– (NSImage *)**image**

Returns the image displayed by the receiver or nil if the receiver is not an image-type cell.

See also: – **setImage:**

imageRectForBounds:

– (NSRect)**imageRectForBounds:**(NSRect)*theRect*

Returns the rectangle that the cell's image is drawn in, which is slightly offset from *theRect*.

See also: – **cellSizeForBounds:**, – **drawingRectForBounds:**

importsGraphics

– (BOOL)**importsGraphics**

Sets whether the text of the receiver (if a text-type cell) is of Rich Text Format (RTF) and thus can import graphics.

See also: – **setImportsGraphics:**

initImageCell:

– (id)**initImageCell:**(NSImage *)*anImage*

Returns an NSCell object initialized with *anImage* and set to have the cell's default menu. If *anImage* is **nil**, no image is set.

initTextCell:

– (id)**initTextCell:**(NSString *)*aString*

Returns an NSCell object initialized with *aString* and set to have the cell's default menu. If no field editor (a shared NSText object) has been created for all NSCells, one is created.

intValue

– (int)**intValue**

Returns the receiver's value as an **int**. If the receiver is not a text-type cell or the cell value is not scannable, the method returns zero.

isBezeled

– (BOOL)**isBezeled**

Returns whether the receiving cell has a bezeled border.

See also: – **setBezeled:**

isBordered

– (BOOL)**isBordered**

Returns whether the receiving cell has a plain border.

See also: – **setBordered:**

isContinuous

– (BOOL)**isContinuous**

Returns whether the receiving cell sends its action message continuously on mouse down.

See also: – **setContinuous:**

isEditable

– (BOOL)**isEditable**

Returns whether the receiving cell is editable.

See also: – **setEditable:**

isEnabled

– (BOOL)**isEnabled**

Returns whether the receiving cell responds to mouse events.

See also: – **setEnabled:**

isEntryAcceptable:

– (BOOL)**isEntryAcceptable:**(NSString *)*aString*

Note: This method is being deprecated in favor of a new class of formatter objects. For more information, see NSFormatter. This documentation is provided only for developers who need to modify older applications.

Returns whether a string representing a numeric or date value (*aString*) is formatted in a way suitable to the entry type.

See also: – **entryType**, – **setEntryType:**

isHighlighted

– (BOOL)**isHighlighted**

Returns whether the receiving cell is highlighted.

isOpaque

– (BOOL)**isOpaque**

Returns whether the receiving cell is opaque (non-transparent).

isScrollable

– (BOOL)**isScrollable**

Returns whether the receiving cell scrolls typed text that exceeds the cell's bounds.

See also: – **setScrollable:**

isSelectable

– (BOOL)**isSelectable**

Returns whether the text of the receiving cell can be selected.

See also: – **setSelectable:**

keyEquivalent

– (NSString *)**keyEquivalent**

Implemented by subclasses to return a key equivalent to clicking the cell. The default implementation returns an empty string object.

menu

– (NSMenu *)**menu**

Returns the menu with commands contextually related to the cell or **nil** if no menu is associated.

See also: – **setMenu:**

menuForEvent:inRect:ofView:

– (NSMenu *)**menuForEvent:**(NSEvent *)*anEvent*
inRect:(NSRect)*cellFrame*
ofView:(NSView *)*aView*

Returns the NSMenu associated with the receiver through the **setMenu:** method and related to *anEvent* when the mouse is detected within *cellFrame*. It is usually invoked by the NSControl (*aView*) managing the receiver. The default implementation simply invokes NSCell's **menu** method and will return **nil** if no menu has been set. Subclasses can override to customize the returned menu according to the event received and the area in which the mouse event occurs.

mnemonic

– (NSString *)**mnemonic**

Returns the character in the cell title that appears underlined for use as a mnemonic. If there is no mnemonic character, returns an empty string.

See also: – **setTitleWithMnemonic:**

mnemonicLocation

– (unsigned int)**mnemonicLocation**

Returns the position of the underlined character in the cell title used as a mnemonic. If there is no mnemonic character, returns NSNotFound.

See also: – **setMnemonicLocation:**

mouseDownFlags

– (int)**mouseDownFlags**

Returns the modifier flags for the last (left) mouse-down event or zero if tracking hasn't occured yet for the cell or if no modifier keys accompanied the mouse-down event.

See also: – **modifierFlags** (NSEvent)

nextState

– (int)**nextState**

Returns the button's next state. If the button has three states, it cycles through them in this order: on, off, mixed, on, and so forth. If the button has two states, it toggles between them.

See also: – **allowsMixedState**, – **setAllowsMixedState:**, – **setNextState**

objectValue

– (id)**objectValue**

Returns the NSCell's value as an Objective-C object if a valid object has been associated with the receiver; otherwise, returns **nil**. To be valid, the cell must have a formatter capable of converting the object to and from its textual representation.

performClick:

– (void)**performClick:**(id)*sender*

Programmatically simulates a mouse click on the receiver, including the invocation of the action method in the target object. Raises an exception if the action message cannot be successfully sent.

refusesFirstResponder

– (BOOL)**refusesFirstResponder**

Returns YES if the cell can ever become the first responder. To find out whether the cell can become first responder at this time, use the method **acceptsFirstResponder**.

See also: – **setRefusesFirstResponder:**

representedObject

– (id)**representedObject**

Returns the object the receiving cell represents. For example, you could have a pop-up list of color names, and the represented objects could be the appropriate NSColor objects.

See also: – **setRepresentedObject:**

resetCursorRect:inView:

– (void)**resetCursorRect:**(NSRect)*cellFrame* **inView:**(NSView *)*controlView*

Sets the receiver to show the I-beam cursor within *cellFrame* while it tracks the mouse . The receiver must be an enabled and selectable (or editable) text-type cell. *controlView* is the NSControl that manages the cell.

selectWithFrame:inView:editor:delegate:start:length:

– (void)**selectWithFrame:**(NSRect)*aRect*
inView:(NSView *)*controlView*
editor:(NSText *)*textObj*
delegate:(id)*anObject*
start:(int)*selStart*
length:(int)*selLength*

Uses the field editor *textObj* to select text in a range marked by *selStart* and *selLength*, which will be highlighted and selected as though the user had dragged the cursor over it. This method is similar to **editWithFrame:inView:editor:delegate:event:**, except that it can be invoked in any situation, not only on a mouse-down event. *aRect* is the rectangle in which the selection should occur, *controlView* is the NSControl managing the receiver, and *anObject* is the delegate of the field editor. Returns without doing anything if *controlView*, *textObj*, or the receiver are **nil**, or if the receiver as no font set for it.

sendActionOn:

– (int)**sendActionOn:**(int)*mask*

Sets the conditions on which the receiver sends action messages to its target and returns a bit mask with which to detect the previous settings. *mask* is set with one or more of these bit masks:

Value	Description
NSLeftMouseUpMask	Don't send action message on (left) mouse up.
NSLeftMouseDownMask	Send action message on (left) mouse down.
NSLeftMouseDraggedMask	Send action message when (left) mouse is dragged.
NSPeriodicMask	Send action message continuously.

You can send **setContinuous:** method to turn on the flag corresponding to NSPeriodicMask or NSLeftMouseDraggedMask, whichever is appropriate to the given subclass of NSCell.

See also: – **action**

sendsActionOnEndEditing

– (BOOL)**sendsActionOnEndEditing**

Returns whether the cell sends its action mesage whenever the user finishes editing the cell's text. If it returns YES, the cell sends its action message when the user does one of the following:

- Presses the Return key
- Deactivates the window containing the cell
- Presses the Tab key to move out of the field
- Clicks on another text field

This behavior is familiar to Mac OS users.

If it returns NO, the cell sends its action message only when the user presses the Return key. This behavior is familiar to OPENSTEP users.

setAction:

– (void)**setAction:**(SEL)*aSelector*

In NSCell, raises NSInternalInconsistencyException. However, NSActionCell overrides this method to set the action method as part of the implementation of the target/action mechanism.

See also: – **action**, – **setTarget:**, – **target**

setAlignment:

– (void)**setAlignment:**(NSTextAlignment)*mode*

Sets the alignment of text in the receiver. *mode* is one of five constants: NSLeftTextAlignment, NSRightTextAlignment, NSCenterTextAlignment, NSJustifiedTextAlignment, NSNaturalTextAlignment (the default alignment for the text).

See also: – **alignment**, – **setWraps:**

setAllowsEditingTextAttributes:

– (void)**setAllowsEditingTextAttributes:**(BOOL)*flag*

Sets whether the textual attributes of the receiver can be modified. If *flag* is NO, the receiver cannot import graphics (that is, it does not support RTFD text).

See also: – **allowsEditingTextAttributes**, – **setImportsGraphics:**

setAllowsMixedState:

– (void)**setAllowsMixedState**:(BOOL)*flag*

If *flag* is YES, the button has three states: on, off, and mixed. If *flag* is NO, the button has two states: on and off.

See also: – **allowsMixedState**, – **nextState**, – **setNextState**

setAttributedStringValue:

– (void)**setAttributedStringValue:**(NSAttributedString *)*attribStr*

Sets the value of the receiver to the attributed string *attribStr*. If a formatter is set for the receiver, but the formatter does not understand the attributed string, it marks *attribStr* as an invalid object. If the receiver is not a text-type cell, it's converted to one. The following example sets the text in a cell to 14 points, red, in the system font.

```
NSColor *txtColor = [NSColor redColor];
NSFont *txtFont = [NSFont boldSystemFontOfSize:14];
NSDictionary *txtDict = [NSDictionary dictionaryWithObjectsAndKeys:txtFont,
    NSFontAttributeName, txtColor, NSForegroundColorAttributeName, nil];
NSAttributedString *attrStr = [[[NSAttributedString alloc]
    initWithString:@"Hello!" attributes:txtDict] autorelease];
[[attrStrTextField cell] setAttributedStringValue:attrStr];
[attrStrTextField updateCell:[attrStrTextField cell]];
```

See also: – **attributedStringValue**

setBezeled:

– (void)**setBezeled:**(BOOL)*flag*

Sets whether the receiver draws itself with a bezeled border. The **setBezeled:** and **setBordered:** methods are mutually exclusive (that is, a border can be only plain or bezeled).

See also: – **isBezeled**

setBordered:

– (void)**setBordered:**(BOOL)*flag*

Sets whether the receiver draws itself outlined with a plain border. The **setBezeled:** and **setBordered:** methods are mutually exclusive (that is, a border can be only plain or bezeled).

See also: – **isBordered**

setCellAttribute:to:

– (void)**setCellAttribute:**(NSCellAttribute)*aParameter* **to:**(int)*value*

Sets a cell attribute identified by *aParameter*—such as the receiver's state, and whether it's disabled, editable, or highlighted—to *value*.

See also: – **cellAttribute:**

setContinuous:

– (void)**setContinuous:**(BOOL)*flag*

Sets whether the receiver continuously sends its action message to its target while it tracks the mouse. In practice, the continuous setting has meaning only for instances of NSActionCell and its subclasses, which

implement the target/action mechanism. Some NSControl subclasses, notably NSMatrix, send a default action to a default target when a cell doesn't provide a target or action.

See also: – **isContinuous**,– **sendActionOn:**

setDoubleValue:

– (void)**setDoubleValue:**(double)*aDouble*

Sets the value of the receiving cell to an object representing a **double**. In its implementation, this method invokes **setObjectValue:**. Does nothing if the receiver is not a text-type cell.

See also: – **doubleValue**

setEditable:

– (void)**setEditable:**(BOOL)*flag*

Sets whether the user can edit the receiver's text. If flag is YES, the text can also be selected. If flag is NO, the selectable attribute is restored to what it was before the cell was last made editable.

See also: – **isEditable**, – **setSelectable:**

setEnabled:

– (void)**setEnabled:**(BOOL)*flag*

Sets whether the receiver is enabled or disabled. The text of disabled cells is changed to gray. If a cell is disabled, it cannot be highlighted, does not support mouse tracking (and thus cannot participate in target/action functionality), and cannot be edited. However, you can still alter many attributes of a disabled cell programmatically (**setState:**, for instance, will still work).

See also: – **isEnabled**

setEntryType:

– (void)**setEntryType:**(int)*aType*

Note: This method is being deprecated in favor of a new class of formatter objects. For more information, see NSFormatter. This documentation is provided only for developers who need to modify older applications.

Sets how numeric data are formatted in the receiver and places restrictions on acceptable input. *aType* can be one of the following constants:

Constant	Restrictions and Other Information
NSIntType	Must be between INT_MIN and INT_MAX
NSPositiveIntType	Must be between 1 and INT_MAX
NSFloatType	Must be between -FLT_MAX and FLT_MAX
NSPositiveFloatType	Must be between FLT_MIN and FLT_MAX
NSDoubleType	Must be between -DBL_MAX and DBL_MAX
NSPositiveDoubleType	Must be between DBL_MAX and DBL_MAX
NSAnyType	Any value is allowed.

If the receiver isn't a text-type cell, this method converts it to one; in the process, it makes its title "Cell" and sets its font to the user's system font at 12 points.

You can check whether formatted strings conform to the entry types of cells with the **isEntryAcceptable:** method. NSControl subclasses also use **isEntryAcceptable:** to validate what users have typed in editable cells. You can control the format of values accepted and displayed in cells by creating a custom subclass of NSFormatter and associating an instance of that class with cells (through **setFormatter:**). In custom NSCell subclasses, you can also override **isEntryAcceptable:** to check for the validity of data entered into cells.

See also: – **entryType**

setFloatingPointFormat:left:right:

– (void)**setFloatingPointFormat:**(BOOL)*autoRange*
left:(unsigned)*leftDigits*
right:(unsigned)*rightDigits*

Note: This method is being deprecated in favor of a new class of formatter objects. For more information, see NSFormatter. This documentation is provided only for developers who need to modify older applications.

Sets whether floating-point numbers are autoranged in the receiver, and sets the sizes of the fields to the left and right of the decimal point. If *autoRange* is NO, *leftDigits* specifies the maximum number of digits to the left of the decimal point, and *rightDigits* specifies the number of digits to the right (the fractional digit places will be padded with zeros to fill this width). However, if a number is too large to fit its integer part

in *leftDigits* digits, as many places as are needed on the left are effectively removed from *rightDigits* when the number is displayed.

If *autoRange* is YES, *leftDigits* and *rightDigits* are simply added to form a maximum total field width for the receiver (plus 1 for the decimal point). The fractional part will be padded with zeros on the right to fill this width, or truncated as much as possible (up to removing the decimal point and displaying the number as an integer). The integer portion of a number is never truncated—that is, it is displayed in full no matter what the field width limit is.

The following example sets a cell used to display dollar amounts up to 99,999.99:

```
[[currencyDollarsField cell] setEntryType:NSFloatType];
[[currencyDollarsField cell] setFloatingPointFormat:NO left:5 right:2];
```

See also: – **setEntryType:**

setFloatValue:

– (void)**setFloatValue:**(float)*aFloat*

Sets the value of the receiving cell to an object representing a **float**. In its implementation, this method invokes **setObjectValue:**. Does nothing if the receiver is not a text-type cell.

See also: – **floatValue**

setFont:

– (void)**setFont:**(NSFont *)*fontObj*

Sets the font to be used when the receiver displays text. If the receiver is not a text-type cell, the method converts it to that type. If *fontObj* is **nil** and the receiver is a text-type cell, the font currently held by the receiver is autoreleased.

See also: – **font**

setFormatter:

– (void)**setFormatter:**(NSFormatter *)*newFormatter*

Sets the formatter object used to format the textual representation of the receiver's object value and to validate cell input and convert it to that object value. If the new formatter cannot interpret the receiver's current object value, that value is converted to a string object. This method retains new formatters and releases replaced ones. If *newFormatter* is **nil**, the receiver is disassociated from the current formatter.

See also: – **formatter**

setImage:

– (void)**setImage:**(NSImage *)*image*

Sets the image to be displayed by the receiver. If the receiver is not an image-type cell, the method converts it to that type. If *image* is **nil** and the receiver is an image-type cell, the image currently held by the receiver is autoreleased.

See also: – **image**

setImportsGraphics:

– (void)**setImportsGraphics:**(BOOL)*flag*

Sets whether the receiver can import images into its text (that is, whether it supports RTFD text). If flag is YES, the receiver is also set to allow editing of text attributes (**setAllowsEditingTextAttributes:**).

See also: – **importsGraphics**

setIntValue:

– (void)**setIntValue:**(int)*anInt*

Sets the value of the receiving cell to an object representing an **int**. In its implementation, this method invokes **setObjectValue:**. Does nothing if the receiver is not a text-type cell.

See also: – **intValue**

setMenu:

– (void)**setMenu:**(NSMenu *)*aMenu*

Associates a menu with the cell that has commands contextually related to the cell (a pop-up menu on Windows). The associated menu is retained. If aMenu is **nil**, any association with a previous menu is removed.

See also: – **menu**

setMnemonicLocation:

– (void)**setMnemonicLocation:**(unsigned int)*location*

Sets the character of the cell title identified by *location* that is to be underlined. This character identifies the access key on Windows by which users can access the cell. *location* must be between 0 and 254.

See also: – **mnemonicLocation**

setNextState

– (void)**setNextState**

Sets the button to its next state. If the button has three states, it cycles through them in this order: on, off, mixed, on, and so forth. If the button has two states, it toggles between them.

See also: – **allowsMixedState**, – **nextState**, – **setAllowsMixedState:**

setObjectValue:

– (void)**setObjectValue:**(id)*object*

Sets the receiver's object value to *object*.

See also: – **objectValue**, – **setRepresentedObject:**

setRefusesFirstResponder:

– (void)**setRefusesFirstResponder**:(BOOL)*flag*

Sets whether the cell can become the first responder. If *flag* is YES, the cell cannot become the first responder.

If **refusesFirstResponder** returns NO and the cell is enabled, the method **acceptsFirstResponder** returns YES, allowing the cell to become first responder

setRepresentedObject:

– (void)**setRepresentedObject:**(id)*anObject*

Sets the object represented by the receiver, for example, an NSColor object for a cell with a title of "Blue."

See also: – **setObjectValue:**, – **representedObject**

setScrollable:

– (void)**setScrollable:**(BOOL)*flag*

Sets whether excess text in the receiver is scrolled past the cell's bounds. If flag is YES, wrapping is turned off. When the scrollable attribute is turned on, the alignment of text in the cell is changed to left alignment.

See also: – **isScrollable**

setSelectable:

– (void)**setSelectable:**(BOOL)*flag*

Sets whether text in the receiver can be selected; always makes the receiver's text uneditable.

See also: – **isSelectable**, – **setEditable:**

setSendsActionOnEndEditing:

– (void)**setSendsActionOnEndEditing:**(BOOL)*flag*;

Sets whether the cell sends its action mesage whenever the user finishes editing the cell's text. If flag is YES, the cell sends its action message when the user does one of the following:

- Presses the Return key
- Deactivates the window containing the cell
- Presses the Tab key to move out of the field
- Clicks on another text field.

This behavior is familiar to Mac OS users.

If *flag* is NO, the cell sends its action message only when the user presses the Return key. This behavior is familiar to OPENSTEP users.

setShowsFirstResponder:

– (void)**setShowsFirstResponder:**(BOOL)*flag*

Sets whether the receiver displays a dotted-line outline when it assumes first responder status.

See also: – **showsFirstResponder**

setState:

– (void)**setState:**(int)*value*

Sets the cell's state to *value*, which can be NSOnState, NSOffState, or NSMixedState.

The cell can have two or three states. If it has two, *value* can be NSOffState (the normal or unpressed state) or NSOnState (the alternate or pressed state). If it has three, *value* can be NSOnState (the feature is in effect everywhere), NSOffState (the feature is in effect nowhere), or NSMixedState (the feature is in effect somewhere). Note that if the cell has only two states and *value* is NSMixedState, this method sets the cell's state to NSOnState.

Although using the enumerated constants is preferred, *value* can also be an integer. If the cell has two states, zero is treated as NSOffState, and a non-zero value is treated as NSOnState. If the cell has three states, zero is treated as NSOffState; a negative value, as NSMixedState; and a positive value, as NSOnState.

Note that the value that **state** returns may not be the same value you passed into **setState:**.

To check whether the button uses the mixed state, use the method **allowsMixedState**.

See also: – **state**

setStringValue:

– (void)**setStringValue:**(NSString *)*aString*

Sets the value of the receiving cell to an NSString object. In its implementation, this method invokes **setObjectValue:**. If no formatter is assigned to the receiver or if the formatter cannot "translate" *aString* to an underlying object, the receiver is flagged as having an invalid object. If the receiver is not a text-type cell, this method converts it to one before setting the object value.

See also: – **stringValue**

setTag:

– (void)**setTag:**(int)*anInt*

Implemented by NSActionCell to set the receiver's tag integer. NSCell's implementation raises NSInternalInconsistencyException.

See also: – **tag**

setTitleWithMnemonic:

– (void)**setTitleWithMnemonic:**(NSString *)*aString*

Sets the title of a cell with a character underlined to denote an access key (Windows only). Use an ampersand character to mark the character (the one following the ampersand) to be underlined. For example, the following message causes the "c" in "Receive" to be underlined:

```
[aCell setTitleWithMnemonic:NSLocalizedString(@"Re&ceive")];
```

See also: – **mnemonic**, – **setMnemonicLocation:**

setTarget:

– (void)**setTarget:**(id)*anObject*

Implemented by NSActionCell to set the receiver's target object receiving the action message. NSCell's implementation raises NSInternalInconsistencyException.

See also: – **target**

setType:

– (void)**setType:**(NSCellType)*aType*

If the type of the receiving cell is different from *aType*, sets it to *aType*, which must one of NSTextCellType, NSImageTypeCell, or NSNullCellType. If *aType* is NSTextCellType, converts the receiver to a cell of that type, giving it a default title and setting the font to the system font at the default size. If *aType* is NSImageTypeCell, sets a **nil** image.

See also: – **type**

setUpFieldEditorAttributes:

– (NSText *)**setUpFieldEditorAttributes:**(NSText *)*textObj*

Sets textual and background attributes of the receiver, depending on certain attributes. If the receiver is disabled, sets the text color to dark gray; otherwise sets it to the default color. If the receiver has a bezeled border, sets the background to the default color for text backgrounds; otherwise, sets it to the color of the receiver's NSControl.

setWraps:

– (void)**setWraps:**(BOOL)*flag*

Sets whether text in the receiver wraps when its length exceeds the frame of the cell. If flag is YES, then it also sets the receiver to be non-scrollable.

See also: – **wraps**

showsFirstResponder

– (BOOL)**showsFirstResponder**

Returns whether the receiver displays a dotted-line outline when it assumes first responder status.

See also: – **setShowsFirstResponder:**

startTrackingAt:inView:

– (BOOL)**startTrackingAt:**(NSPoint)*startPoint* **inView:**(NSView *)*controlView*

NSCell's implementation of **trackMouse:inRect:ofView:untilMouseIsUp:** invokes this method when tracking begins. *startPoint* is the point the mouse is currently at and *controlView* is the NSControl managing the receiver. NSCell's default implementation returns YES if the receiver is set to respond continuously or when the mouse is dragged. Subclasses override this method to implement special mouse-tracking behavior at the beginning of mouse tracking, for example, displaying a special cursor.

See also: – **continueTracking:at:inView:**, – **stopTracking:at:inView:mouseIsUp:**

state

– (int)**state**

Returns the button's state. The cell can have two or three states. If it has two, it returns either NSOffState (the normal or unpressed state) or NSOnState (the alternate or pressed state). If it has three, it returns NSOnState (the feature is in effect everywhere), NSOffState (the feature is in effect nowhere), or NSMixedState (the feature is in effect somewhere).

To check whether the button uses the mixed state, use the method **allowsMixedState**.

Note that the value that **state** returns may not be the same value you passed into **setState:**.

See also: – **setState:**

stopTracking:at:inView:mouseIsUp:

– (void)**stopTracking:**(NSPoint)*lastPoint*
at:(NSPoint)*stopPoint*
inView:(NSView *)*controlView*
mouseIsUp:(BOOL)*flag*

NSCell's implementation of **trackMouse:inRect:ofView:untilMouseIsUp:** invokes this method when the mouse has left the bounds of the receiver or the mouse goes up (in which case *flag* is YES). *lastPoint* is the point the mouse was at and *stopPoint* is its current point. *controlView* is the NSControl managing the receiver. NSCell's default implementation does nothing. Subclasses often override this method to provide customized tracking behavior. The following example increments the state of a tri-state cell when the mouse is clicked.

```
- (void)stopTracking:(NSPoint)lastPoint at:(NSPoint)stopPoint
    inView:(NSView *)controlView mouseIsUp:(BOOL)flag
{
    if (flag == YES) {
        [self setTriState:([self triState]+1)];
    }
}
```

See also: – **startTrackingAt:inView:**, – **stopTracking:at:inView:mouseIsUp:**

stringValue

– (NSString *)**stringValue**

Returns the receiver's value as an NSString as converted by the receiver's formatter, if one exists. If no formatter exists and the value is an NSString, returns the value as an plain, attributed or localized formatted string. If the value is not an NSString or can't be converted to one, returns an empty string.

See also: – **setStringValue:**

tag

– (int)**tag**

Implemented by NSActionCell to return the receiver's tag integer. NSCell's implementation returns -1.

See also: – **setTag:**

takeDoubleValueFrom:

– (void)**takeDoubleValueFrom:**(id)*sender*

Sets the receiver's own value as a **double** using the **double** value of *sender*.

See also: – **setDoubleValue:**

takeFloatValueFrom:

– (void)**takeFloatValueFrom:**(id)*sender*

Sets the receiver's own value as a **float** using the **float** value of *sender*.

See also: – **setFloatValue:**

takeIntValueFrom:

– (void)**takeIntValueFrom:**(id)*sender*

Sets the receiver's own value as an **int** using the **int** value of *sender*. The following example shows this method being used to write the value taken from a slider (**sender**) to a text field cell:

```
- (void)sliderMoved:(id)sender
{
    [[valueField cell] takeIntValueFrom:[sender cell]];
    [valueField display];
}
```

See also: – **setIntValue:**

takeObjectValueFrom:

– (void)**takeObjectValueFrom:**(id)*sender*

Sets the receiver's own value as an object using the object value of *sender*.

See also: – **setObjectValue:**

takeStringValueFrom:

– (void)**takeStringValueFrom:**(id)*sender*

Sets the receiver's own value as a string object using the NSString value of *sender*.

See also: – **setStringValue:**

target

– (id)**target**

Implemented by NSActionCell to return the target object to which the receiver's action message is sent. NSCell's implementation returns **nil**.

See also: – **setTarget:**

titleRectForBounds:

– (NSRect)**titleRectForBounds:**(NSRect)*theRect*

If the receiver is a text-type cell, resizes the drawing rectangle for the title (*theRect*) inward by a small offset to accommodate the cell border. If the receiver is not a text-type cell, the method does nothing.

See also: – **imageRectForBounds:**

trackMouse:inRect:ofView:untilMouseUp:

– (BOOL)**trackMouse:**(NSEvent *)*theEvent*
inRect:(NSRect)*cellFrame*
ofView:(NSView *)*controlView*
untilMouseUp:(BOOL)*flag*

Invoked by an NSControl to initiate the tracking behavior of one of its NSCells. It's generally not overriden since the default implementation invokes other NSCell methods that can be overriden to handle specific events in a dragging session. This method's return value depends on the *untilMouseUp* flag. If that flag is set to YES, this method returns YES if the mouse goes up anywhere; NO, otherwise. If that flag is set to NO, this method returns YES if the mouse goes up within *cellFrame*; NO, otherwise. The argument *theEvent* is typically the mouse event received by the initiating NSControl, usually identified by *controlView*. The *flag* argument indicates whether tracking should continue until the mouse button goes up; if *flag* is NO, tracking ends when the mouse is dragged after the initial mouse down.

This method first invokes **startTrackingAt:inView:**. If that method returns YES, then as mouse-dragged events are intercepted, **continueTracking:at:inView:** is invoked. Finally, s**topTracking:at:inView: mouseIsUp:** is invoked. If *untilMouseUp* is YES, it's invoked when the mouse goes up anywhere, If *untilMouseUp* is NO, it's invoked when the mouse goes up within *cellFrame*. (If *cellFrame* is NULL, then the bounds are considered infinitely large.) You usually override one or more of these methods to respond to specific mouse events.

type

– (NSCellType)**type**

Returns the type of the receiver, one of NSTextCellType, NSImageTypeCell, or NSNullCellType.

See also: – **setType:**

wraps

– (BOOL)**wraps**

Returns whether text of the receiver wraps when it exceeds the borders of the cell.

See also: – **setWraps:**

NSClipView

Inherits From: NSView : NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSClipView.h

Class at a Glance

Purpose

An NSClipView contains and scrolls the document view displayed by an NSScrollView. You normally don't need to program with NSClipViews, as NSScrollView handles most of the details of their operation.

Principal Attributes

- Efficient scrolling by copying drawn portions of the document view
- Monitoring of document view for automatic update

Creation

Interface Builder	
– initWithFrame:	Initializes the NSClipView.

Commonly Used Methods

– setDocumentView:	Sets the view scrolled within the NSClipView.
– setCopiesOnScroll:	Sets whether the NSClipView copies drawn portions of the document view during scrolling.

Class Description

An NSClipView holds the document view of an NSScrollView, clipping the document view to its frame, handling the details of scrolling in an efficient manner, and updating the NSScrollView when the document view's size or position changes. You don't normally use the NSClipView class directly; it's provided primarily as the scrolling machinery for the NSScrollView class. However, you might use the NSClipView class to implement a class similar to NSScrollView.

When an NSClipView is instructed to scroll its document view, it copies as much of the already-drawn document view as possible. This allows for efficient scrolling by obviating the need to redraw large portions of the document view. The NSClipView then sends its document view a **setNeedsDisplayInRect:** message to mark as invalid the newly exposed region(s) of the document view. If copying drawn areas is inappropriate for your needs, you can turn it off by sending the NSClipView a **setCopiesOnScroll:** message with an argument of NO.

In addition to performing the details of scrolling, an NSClipView monitors its document view and sends its superview (usually an NSScrollView) a **reflectScrolledClipView:** message whenever the relationship between the NSClipView and the document view has changed. This allows the superview to update itself to reflect the change—for example, an NSScrollView uses this method to change the position of its scrollers when the user causes the document view to autoscroll or when the document view's size changes.

Method Types

Setting the document view

– setDocumentView:
– documentView

Scrolling

– scrollToPoint:
– autoscroll:
– constrainScrollPoint:

Determining scrolling efficiency

– setCopiesOnScroll:
– copiesOnScroll

Getting the visible portion

– documentRect
– documentVisibleRect

Setting the document cursor

– setDocumentCursor:
– documentCursor

Setting the background color

– setBackgroundColor:
– backgroundColor

Overridden NSView methods

– acceptsFirstResponder
– becomeFirstResponder
– isFlipped
– rotateByAngle:
– scaleUnitSquareToSize:
– setBoundsOrigin:
– setBoundsRotation:
– setBoundsSize:
– setFrameSize:
– setFrameOrigin:
– setFrameRotation:
– setNextKeyView:
– translateOriginToPoint:
– viewBoundsChanged:
– viewFrameChanged:

Instance Methods

acceptsFirstResponder

– (BOOL)**acceptsFirstResponder**

If the receiver has a document view, returns the document view's **acceptsFirstResponder**. Otherwise returns NO.

See also: – **documentView**, – **acceptsFirstResponder** (NSResponder)

autoscroll:

– (BOOL)**autoscroll:**(NSEvent *)*theEvent*

Scrolls the receiver proportionally to *theEvent*'s distance outside of it. *theEvent*'s location should be expressed in the window's base coordinate system (which it normally is), not the receiving NSClipView's. Returns YES if any scrolling is performed; otherwise returns NO.

Never invoke this method directly; instead, the NSClipView's document view should repeatedly send itself **autoscroll:** messages when the mouse is dragged outside the NSClipView's frame during a modal event loop initiated by a mouse-down event. The NSView class implements **autoscroll:** to forward the message to the receiver's superview; thus the message is ultimately forwarded to the NSClipView.

backgroundColor

– (NSColor *)**backgroundColor**

Returns the color of the receiver's background.

See also: – **setBackgroundColor:**

becomeFirstResponder

– (BOOL)**becomeFirstResponder**

If the key view selection direction of the receiver's NSWindow isn't NSSelectingPrevious, attempts to make the document view the first responder. If the direction is NSSelectingPrevious, attempts to make the receiver's previous key view (typically the containing NSScrollView) the first responder. Returns YES if successful and NO otherwise

See also: – **becomeFirstResponder** (NSResponder), – **makeFirstResponder:** (NSWindow), – **keyViewSelectionDirection** (NSWindow)

constrainScrollPoint:

– (NSPoint)**constrainScrollPoint:**(NSPoint)*proposedNewOrigin*

Returns a scroll point adjusted from *proposedNewOrigin*, if necessary, to guarantee that the receiver will still lie within its document view. For example, if *proposedNewOrigin*'s *y* coordinate lies to the left of the document view's origin, then the *y* coordinate returned is set to that of the document view's origin.

See also: – **scrollToPoint:**

copiesOnScroll

– (BOOL)**copiesOnScroll**

Returns YES if the receiver copies its existing rendered image while scrolling (only drawing exposed portions of its document view), NO if it forces its contents to be redrawn each time.

See also: – **setCopiesOnScroll:**

documentCursor

– (NSCursor *)**documentCursor**

Returns the cursor object used when the mouse lies over the receiver.

See also: – **setDocumentCursor:**

documentRect

– (NSRect)**documentRect**

Returns the rectangle defining the document view's frame, adjusted to the size of the receiver if the document view is smaller. In other words, this rectangle is always at least as large as the receiver itself.

The document rectangle is used in conjunction with an NSClipView's bounds rectangle to determine values for the indicators of relative position and size between the NSClipView and its document view. For example, NSScrollView uses these rectangles to set the size and position of the knobs in its scrollers. When the document view is much larger than the NSClipView, the knob is small; when the document view is near the same size, the knob is large; and when the document view is the same size or smaller, there is no knob.

See also: – **reflectScrolledClipView:** (NSScrollView), – **documentVisibleRect**

documentView

– (id)**documentView**

Returns the receiver's document view.

See also: – **setDocumentView:**

documentVisibleRect

– (NSRect)**documentVisibleRect**

Returns the exposed rectangle of the receiver's document view, in the document view's own coordinate system. Note that this rectangle doesn't reflect the effects of any clipping that may occur above the NSClipView itself. To get the portion of the document view that's guaranteed to be visible, send it a **visibleRect** message.

See also: – **documentRect**

isFlipped

– (BOOL)**isFlipped**

Returns YES if the document view is flipped, NO if it isn't.

See also: – **isFlipped** (NSView)

rotateByAngle:

– (void)**rotateByAngle:**(float)*angle*

Overrides NSView's implementation to disable rotation.

scaleUnitSquareToSize:

– (void)**scaleUnitSquareToSize:**(NSSize)*newUnitSize*

Performs as NSView's implementation and updates a containing NSScrollView based on the new bounds.

scrollToPoint:

– (void)**scrollToPoint:**(NSPoint)*newOrigin*

Changes the origin of the receiver's bounds rectangle to *newOrigin*.

See also: – **constrainScrollPoint:**

setBackgroundColor:

– (void)**setBackgroundColor:**(NSColor *)*aColor*

Sets the receiver's background color to *aColor*.

See also: – **backgroundColor**

setBoundsOrigin:

– (void)**setBoundsOrigin:**(NSPoint)*aPoint*

Performs as NSView's implementation and updates a containing NSScrollView based on the new bounds.

setBoundsRotation:

– (void)**setBoundsRotation:**(float)*angle*

Overrides NSView's implementation to disable rotation.

setBoundsSize:

– (void)**setBoundsSize:**(NSSize)*aSize*

Performs as NSView's implementation and updates a containing NSScrollView based on the new bounds.

setCopiesOnScroll:

– (void)**setCopiesOnScroll:**(BOOL)*flag*

Controls whether the receiver copies rendered images while scrolling. If *flag* is YES, the receiver copies the existing rendered image to its new location while scrolling, and only draws exposed portions of its document view. If *flag* is NO, the receiver always forces its document view to draw itself on scrolling.

See also: – **copiesOnScroll**

setDocumentCursor:

– (void)**setDocumentCursor:**(NSCursor *)*aCursor*

Sets the cursor object used over the receiver to *aCursor*.

See also: – **documentCursor**

setDocumentView:

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

Sets the receiver's document view to *aView*, removing any previous document view, and sets the origin of the receiver's bounds rectangle to the origin of *aView*'s frame rectangle. If the receiver is contained in an NSScrollView, you should send the NSScrollView a **setDocumentView:** message instead, so that it can perform whatever updating it needs.

In the process of setting the document view, this method registers the receiver for the notifications NSViewFrameDidChangeNotification and NSViewBoundsDidChangeNotification, adjusts the key view loop to include the new document view, and updates a parent NSScrollView's display if needed using **reflectScrolledClipView:**.

See also: – **documentView**

setFrameOrigin:

– (void)**setFrameOrigin:**(NSPoint)*aPoint*

Performs as NSView's implementation and updates a containing NSScrollView based on the new bounds.

setFrameRotation:

– (void)**setFrameRotation:**(float)*angle*

Overrides NSView's implementation to disable rotation.

setFrameSize:

– (void)**setFrameSize:**(NSSize)*aSize*

Performs as NSView's implementation and updates a containing NSScrollView based on the new bounds.

setNextKeyView:

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

Performs as NSView's implementation, except inserts the receiver's document view between itself and *aView* in the key view loop.

See also: – **setNextKeyView:** (NSView)

translateOriginToPoint:

– (void)**translateOriginToPoint:**(NSPoint)*aPoint*

Performs as NSView's implementation and updates a containing NSScrollView based on the new bounds.

viewBoundsChanged:

– (void)**viewBoundsChanged:**(NSNotification *)*aNotification*

Handles an NSViewBoundsDidChangeNotification by updating a containing NSScrollView based on the new bounds.

viewFrameChanged:

– (void)**viewFrameChanged:**(NSNotification *)*aNotification*

Handles an NSViewFrameDidChangeNotification by updating a containing NSScrollView based on the new frame.

NSCoderAdditions

Inherits From: NSObject

Declared In: AppKit/NSColor.h

Class Description

This category adds a single method to the Foundation framework's NSCoder class. This method, **decodeNXColor**, is used to convert archived NXColors into NSColors.

NXColor, a type that dates from pre-OpenStep versions of NEXTSTEP, was a **struct**. Its replacement, NSColor, is a class. The difficulties of converting from a struct to a class require a special method like **decodeNXColor**.

Note: **decodeNXColor** becomes part of the NSCoder class only for applications that use the Application Kit.

For more information, see the NSCoder class specification in the *Foundation Framework Reference*.

Instance Methods

decodeNXColor

– (NSColor *)**decodeNXColor**

Decodes a color object from NEXTSTEP Release 3 or earlier and returns an autoreleased NSColor object. This method does not have a matching method for encoding an NXColor object. Encode an NSColor object instead.

NSColor

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

Class at a Glance

Purpose

An NSColor object represents a color, which is defined in a *color space*, each point of which has a set of components (such as red, green, and blue) that uniquely define a color.

Principal Attributes

- Color space
- Color components

Creation

Various **colorWith...** and **colorUsing...** methods
Preset colors: **blackColor**, **blueColor**, etc.

Commonly Used Methods

colorUsingColorSpaceName:	Creates an NSColor in the specified color space.
set	Sets the drawing color.

Class Description

An NSColor object represents color and sometimes opacity (alpha). By sending a **set** message to an NSColor instance, you set the color for the current PostScript drawing context. This causes subsequently drawn graphics to have the color represented by the NSColor instance.

Color Spaces

A color is defined in some particular *color space*. A color space consists of a set of dimensions—such as red, green, and blue in the case of RGB space. Each point in the space represents a unique color, and the point's location along each dimension is called a *component*. An individual color is usually specified by the numeric values of its components, which range from 0.0 to 1.0. For instance, a pure red is specified in RGB space by the component values 1.0, 0.0, and 0.0.

Some color spaces include an alpha component, which defines the color's opacity. An alpha value of 1.0 means completely opaque, and 0.0 means completely transparent. The alpha component is ignored when the color is used on a device that doesn't support alpha, such as a printer.

There are three kinds of color spaces in the Application Kit:

- *Device-dependent*. This means that a given color might not look the same on different displays and printers. The components of a device-dependent color correspond to the inks in a printer or the electron guns in a monitor. Because printer inks and screen phosphors vary from device to device, you can't expect a consistent color from a device-dependent color space.
- *Device-independent*, also known as *calibrated*. With this sort of color space, a given color should look the same on all devices.
- *Named*. The "named color space" has components that aren't numeric values, but simply names in various catalogs of colors. Named colors come with lookup tables that provide the ability to generate the correct color on a given device.

NSColors provided by the Application Kit use eight different color spaces, referred to by these global NSString variables:

Global Variable	Color Space Description
NSDeviceCMYKColorSpace	Cyan, magenta, yellow, black, and alpha components
NSDeviceWhiteColorSpace	White and alpha components
NSDeviceBlackColorSpace	Black and alpha components
NSDeviceRGBColorSpace	Red, green, blue, and alpha components Hue, saturation, brightness, and alpha components
NSCalibratedWhiteColorSpace	White and alpha components

Global Variable	Color Space Description
NSCalibratedBlackColorSpace	Black and alpha components
NSCalibratedRGBColorSpace	Red, green, blue, and alpha components Hue, saturation, brightness, and alpha components
NSNamedColorSpace	Catalog name and color name components

Color spaces whose names start with “NSDevice” are device-dependent; those whose names start with “NSCalibrated” are device-independent.

Color Components

There’s usually no need to retrieve the individual components of a color, but when needed, you can either retrieve a set of components (using such methods as **getRed:green:blue:alpha:**) or an individual component (using such methods as **redComponent**). However, it’s illegal to ask an NSColor for components that aren’t defined for its color space. You can identify the color space by sending a **colorSpaceName** message to the NSColor object. If you need to ask an NSColor for components that aren’t in its color space (for instance, when you’ve gotten the color from the color panel), first convert the color to the appropriate color space using the **colorUsingColorSpaceName:** method. If the color is already in the specified color space, you get the same color back; otherwise you get a conversion that’s usually lossy or that’s correct only for the current device. You get back **nil** if the specified conversion can’t be done.

Creating Subclasses

Subclasses of NSColor need to implement the **colorSpaceName** and **set** methods, as well as the methods that return the components for that color space and the methods in the NSCoding protocol. Some other methods—such as **colorWithAlphaComponent:**, **isEqual:**, and **colorUsingColorSpaceName:device:**—may also be implemented if they make sense for the color space. Mutable subclasses (if any) should additionally implement **copyWithZone:** to provide a true copy.

System Colors

NSColor has a number of methods which return “system” colors; colors that are controlled by user preferences. These colors—including **controlColor**, **textColor**, and **selectedTextColor**—should be used by developers who want to create custom controls or subclass existing controls which honor the user's color preferences. System colors are implemented as named colors in a special color list named “System.” To extract the components of a system color, you must use NSColor's **colorUsingColorSpaceName:** method to convert the color to a color space known to respond to the component accessor methods you need.

An NSSystemColorsDidChangeNotification is sent when the system colors have been changed (such as through a system control panel interface). If you have any non-system colors that depend on the system colors, you can change them when you receive this notification.

Adopted Protocols

NSCoding

– encodeWithCoder:
– initWithCoder:

NSCopying

– copyWithZone:

Method Types

Creating an NSColor object from Component Values

+ colorWithCalibratedHue:saturation:brightness:alpha:
+ colorWithCalibratedRed:green:blue:alpha:
+ colorWithCalibratedWhite:alpha:
+ colorWithCatalogName:colorName:
+ colorWithDeviceCyan:magenta:yellow:black:alpha:
+ colorWithDeviceHue:saturation:brightness:alpha:
+ colorWithDeviceRed:green:blue:alpha:
+ colorWithDeviceWhite:alpha:

Creating an NSColor With Preset Components

+ blackColor
+ blueColor
+ brownColor
+ clearColor
+ cyanColor
+ darkGrayColor
+ grayColor
+ greenColor
+ lightGrayColor
+ magentaColor
+ orangeColor
+ purpleColor
+ redColor
+ whiteColor
+ yellowColor

Creating a system color—that is, an NSColor whose value is specified by user preferences

+ controlBackgroundColor
+ controlColor
+ controlHighlightColor
+ controlLightHighlightColor
+ controlShadowColor
+ controlDarkShadowColor
+ controlTextColor
+ disabledControlTextColor
+ gridColor
+ highlightColor
+ knobColor
+ scrollBarColor
+ selectedControlColor
+ selectedControlTextColor
+ selectedMenuItemColor
+ selectedMenuItemTextColor
+ selectedTextBackgroundColor
+ selectedTextColor
+ selectedKnobColor
+ shadowColor
+ textBackgroundColor
+ textColor
+ windowFrameColor
+ windowFrameTextColor

Ignoring Alpha Components

+ ignoresAlpha
+ setIgnoresAlpha:

Copying and Pasting

+ colorFromPasteboard:
– writeToPasteboard:

Retrieving a Set of Components

– getCyan:magenta:yellow:black:alpha:
– getHue:saturation:brightness:alpha:
– getRed:green:blue:alpha:
– getWhite:alpha:

Retrieving Individual Components

– alphaComponent
– blackComponent
– blueComponent
– brightnessComponent
– catalogNameComponent
– colorNameComponent
– cyanComponent
– greenComponent
– hueComponent
– localizedCatalogNameComponent
– localizedColorNameComponent
– magentaComponent
– redComponent
– saturationComponent
– whiteComponent
– yellowComponent

Converting to Another Color Space

– colorSpaceName
– colorUsingColorSpaceName:
– colorUsingColorSpaceName:device:

Changing the Color

– blendedColorWithFraction:ofColor:
– colorWithAlphaComponent:
– highlightWithLevel:
– shadowWithLevel:

Drawing

– drawSwatchInRect:
– set

Class Methods

blackColor

+ (NSColor *)**blackColor**

Returns an NSColor in NSCalibratedWhiteColorSpace whose grayscale value is 0.0 and whose alpha value is 1.0.

See also: **– blackComponent**

blueColor

+ (NSColor *)**blueColor**

Returns an NSColor in NSCalibratedRGBColorSpace whose RGB value is 0.0, 0.0, 1.0 and whose alpha value is 1.0.

See also: **– blueComponent**

brownColor

+ (NSColor *)**brownColor**

Returns an NSColor in NSCalibratedRGBColorSpace whose RGB value is 0.6, 0.4, 0.2 and whose alpha value is 1.0.

clearColor

+ (NSColor *)**clearColor**

Returns an NSColor in NSCalibratedWhiteColorSpace whose grayscale and alpha values are both 0.0.

colorFromPasteboard:

+ (NSColor *)**colorFromPasteboard:**(NSPasteboard *)*pasteBoard*

Returns the NSColor currently on the pasteboard, or **nil** if the pasteboard doesn't contain color data. The returned color's alpha component is set to 1.0 if **ignoresAlpha** returns YES.

See also: **– writeToPasteboard:**

colorWithCalibratedHue:saturation:brightness:alpha:

+ (NSColor *)**colorWithCalibratedHue:**(float)*hue*
saturation:(float)*saturation*
brightness:(float)*brightness*
alpha:(float)*alpha*

Creates and returns an NSColor whose color space is NSCalibratedRGBColorSpace, whose opacity value is *alpha*, and whose components in HSB space would be *hue*, *saturation*, and *brightness*. (Values below 0.0 are interpreted as 0.0, and values above 1.0 are interpreted as 1.0.)

See also: **+ colorWithCalibratedRed:green:blue:alpha:**, **+ colorWithDeviceHue:saturation:brightness:alpha:**, **– getHue:saturation:brightness:alpha:**

colorWithCalibratedRed:green:blue:alpha:

+ (NSColor *)**colorWithCalibratedRed:**(float)*red*
green:(float)*green*
blue:(float)*blue*
alpha:(float)*alpha*

Creates and returns an NSColor whose color space is NSCalibratedRGBColorSpace, whose opacity value is *alpha*, and whose RGB components are *red*, *green*, and *blue*. (Values below 0.0 are interpreted as 0.0, and values above 1.0 are interpreted as 1.0.)

See also: **+ colorWithCalibratedHue:saturation:brightness:alpha:**, **+ colorWithDeviceRed:green:blue:alpha:**, **– getRed:green:blue:alpha:**

colorWithCalibratedWhite:alpha:

+ (NSColor *)**colorWithCalibratedWhite:**(float)*white*
alpha:(float)*alpha*

Creates and returns an NSColor whose color space is NSCalibratedWhiteColorSpace, whose opacity value is *alpha*, and whose grayscale value is *white*. (Values below 0.0 are interpreted as 0.0, and values above 1.0 are interpreted as 1.0.)

See also: **+ colorWithDeviceWhite:alpha:**, **– getWhite:alpha:**

colorWithCatalogName:colorName:

+ (NSColor *)**colorWithCatalogName:**(NSString *)*listName*
colorName:(NSString *)*colorName*

Creates and returns an NSColor whose color space is NSNamedColorSpace, by finding the color named *colorName* in the catalog named *listName*, which may be a standard catalog.

See also: **– catalogNameComponent**, **– colorNameComponent**, **– localizedCatalogNameComponent**

colorWithDeviceCyan:magenta:yellow:black:alpha:

+ (NSColor *)**colorWithDeviceCyan:**(float)*cyan*
magenta:(float)*magenta*
yellow:(float)*yellow*
black:(float)*black*
alpha:(float)*alpha*

Creates and returns an NSColor whose color space is NSDeviceCMYKColorSpace, whose opacity value is *alpha*, and whose CMYK components are *cyan*, *magenta*, *yellow*, and *black*. (Values below 0.0 are

interpreted as 0.0, and values above 1.0 are interpreted as 1.0.) In PostScript, this colorspace corresponds directly to the device-dependent operator *setcmykcolor.*

See also: **– getCyan:magenta:yellow:black:alpha:**

colorWithDeviceHue:saturation:brightness:alpha:

+ (NSColor *)**colorWithDeviceHue:**(float)*hue*
saturation:(float)*saturation*
brightness:(float)*brightness*
alpha:(float)*alpha*

Creates and returns an NSColor whose color space is NSDeviceRGBColorSpace, whose opacity value is *alpha*, and whose components in HSB space would be *hue*, *saturation*, and *brightness*. (Values below 0.0 are interpreted as 0.0, and values above 1.0 are interpreted as 1.0.) In PostScript, this colorspace corresponds directly to the device-dependent operator *setrgbcolor.*

See also: **+ colorWithCalibratedHue:saturation:brightness:alpha:**, **+ colorWithDeviceRed:green:blue:alpha:**, **– getHue:saturation:brightness:alpha:**

colorWithDeviceRed:green:blue:alpha:

+ (NSColor *)**colorWithDeviceRed:**(float)*red*
green:(float)*green*
blue:(float)*blue*
alpha:(float)*alpha*

Creates and returns an NSColor whose color space is NSDeviceRGBColorSpace, whose opacity value is *alpha*, and whose RGB components are *red*, *green*, and *blue*. (Values below 0.0 are interpreted as 0.0, and values above 1.0 are interpreted as 1.0.) In PostScript, this colorspace corresponds directly to the device-dependent operator *setrgbcolor.*

See also: **+ colorWithCalibratedRed:green:blue:alpha:**, **+ colorWithDeviceHue:saturation:brightness:alpha:**, **– getRed:green:blue:alpha:**

colorWithDeviceWhite:alpha:

+ (NSColor *)**colorWithDeviceWhite:**(float)*white*
alpha:(float)*alpha*

Creates and returns an NSColor whose color space is NSDeviceWhiteColorSpace, whose opacity value is *alpha*, and whose grayscale value is *white*. (Values below 0.0 are interpreted as 0.0, and values above 1.0

are interpreted as 1.0.) In PostScript, this colorspace corresponds directly to the device-dependent operator *setgray.*

See also: **+ colorWithCalibratedWhite:alpha:**, **– getWhite:alpha:**

controlBackgroundColor

+ (NSColor *)**controlBackgroundColor**

Returns the system color used for the background of large controls such as browsers, table views and clip views. By default, this color is light gray on the Macintosh, and COLOR_WINDOW on Windows. For general information on system colors, see the “System Colors” section of the class description, above.

controlColor

+ (NSColor *)**controlColor**

Returns the system color used for the flat surfaces of a control. By default, the control color is light gray on the Macintosh, and COLOR_3DFACE in Windows. A control’s beveled edges, which set it in relief, are drawn in the colors returned by **controlShadowColor**, **controlDarkShadowColor**, **controlHighlightColor** and **controlLightHighlightColor**. When a control is selected—that is, clicked or dragged—it changes to the color returned by **selectedControlColor**.

The return value of **controlColor** is also the system color used for window backgrounds. For general information about system colors, see the “System Colors” section of the class description, above.

controlDarkShadowColor

+ (NSColor *)**controlDarkShadowColor**

Returns the system color used for the dark edge of the shadow dropped from controls. Controls are displayed as though they were lit from the upper left. Two dark borders, representing shadows, run along the bottom and right. **controlDarkShadowColor** returns the color of the outer, darker border. By default, this color is black on the Macintosh, and COLOR_3DDKSHADOW on Windows. For general information about system colors, see the “System Colors” section of the class description, above.

See also: + **controlShadowColor**

controlHighlightColor

+ (NSColor *)**controlHighlightColor**

Returns the system color used for the highlighted bezels of controls. Controls are displayed as though they were lit from the upper left. Two light borders, representing reflections from the light source, run along the

top and left. **controlHighlightColor** returns the color of the inner, duller border. By default, this color is light gray on the Macintosh, and COLOR_3DLIGHT on Windows. For general information about system colors, see the “System Colors” section of the class description, above.

See also: + **controlLightHighlightColor**

controlLightHighlightColor

+ (NSColor *)**controlLightHighlightColor**

Returns the system color used for light highlights in controls. Controls are displayed as though they were lit from the upper left. Two light borders, representing reflections from the light source, run along the top and left. **controlLightHighlightColor** returns the color of the outer, brighter border. By default, this color is white on the Macintosh, and COLOR_3DHILIGHT on Windows. For general information about system colors, see the “System Colors” section of the class description, above.

See also: + **controlHighlightColor**

controlShadowColor

+ (NSColor *)**controlShadowColor**

Returns the system color used for the shadows dropped from controls. Controls are displayed as though they were lit from the upper left. Two dark borders, representing shadows, run along the bottom and right. **controlShadowColor** returns the color of the inner, lighter border. By default, this color is dark gray on the Macintosh, and COLOR_3DSHADOW on Windows. For general information about system colors, see the “System Colors” section of the class description, above.

See also: + **controlDarkShadowColor**

controlTextColor

+ (NSColor *)**controlTextColor**

Returns the system color used for text on controls that aren’t disabled. By default, the text color is black on the Macintosh, and COLOR_BTNTEXT on Windows. For general information about system colors, see the “System Colors” section of the class description, above.

See also: + **disabledControlTextColor**

cyanColor

+ (NSColor *)**cyanColor**

Returns an NSColor in NSCalibratedRGBColorSpace whose RGB value is 0.0, 1.0, 1.0 and whose alpha value is 1.0.

See also: **– cyanComponent**

darkGrayColor

+ (NSColor *)**darkGrayColor**

Returns an NSColor in NSCalibratedWhiteColorSpace whose grayscale value is 1/3 and whose alpha value is 1.0.

See also: + **lightGrayColor**, + **grayColor**

disabledControlTextColor

+ (NSColor *)**disabledControlTextColor**

Returns the system color used for text on disabled controls. By default, the text color is dark gray on the Macintosh, and COLOR_3DSHADOW on Windows. For general information about system colors, see the "System Colors" section of the class description, above.

See also: + **controlTextColor**

grayColor

+ (NSColor *)**grayColor**

Returns an NSColor in NSCalibratedWhiteColorSpace whose grayscale value is 0.5 and whose alpha value is 1.0.

See also: + **lightGrayColor**, + **darkGrayColor**

greenColor

+ (NSColor *)**greenColor**

Returns an NSColor in NSCalibratedRGBColorSpace whose RGB value is 0.0, 1.0, 0.0 and whose alpha value is 1.0.

See also: **– greenComponent**

gridColor

+ (NSColor *)**gridColor**

Returns the system color used for the optional gridlines in, for example, a table view. By default, this color is gray on the Macintosh, and COLOR_3DFACE on Windows. For general information about system colors, see the “System Colors” section of the class description, above.

highlightColor

+ (NSColor *)**highlightColor**

Returns the system color that represents the virtual light source on the screen. By default, this color is white on the Macintosh, and COLOR_3DHILIGHT on Windows. This method is invoked by the **highlightWithLevel:** method. For general information about system colors, see the “System Colors” section of the class description, above.

See also: **– highlightWithLevel:**

knobColor

+ (NSColor *)**knobColor**

Returns the system color used for the flat surface of a slider knob that hasn’t been selected. By default, this color is light blue on the Macintosh, and COLOR_3DFACE on Windows. The knob’s beveled edges, which set it in relief, are drawn in highlighted and shadowed versions of the face color. When a knob is selected, its color changes to **selectedKnobColor**. For general information about system colors, see the “System Colors” section of the class description, above.

ignoresAlpha

+ (BOOL)**ignoresAlpha**

Returns YES if the application doesn’t support alpha. This value returned is consulted when an application imports alpha (through color dragging, for instance). The value determines whether the color panel has an opacity slider. This value is YES by default, indicating that the opacity components of imported colors will be set to 1.0. If an application wants alpha, it can invoke the **setIgnoresAlpha:** method with a parameter of NO.

See also: **+ setIgnoresAlpha:**, **– alphaComponent**

lightGrayColor

+ (NSColor *)**lightGrayColor**

Returns an NSColor in NSCalibratedWhiteColorSpace whose grayscale value is 2/3 and whose alpha value is 1.0.

See also: + **grayColor**, + **darkGrayColor**

magentaColor

+ (NSColor *)**magentaColor**

Returns an NSColor in NSCalibratedRGBColorSpace whose RGB value is 1.0, 0.0, 1.0 and whose alpha value is 1.0.

See also: – **magentaComponent**

orangeColor

+ (NSColor *)**orangeColor**

Returns an NSColor in NSCalibratedRGBColorSpace whose RGB value is 1.0, 0.5, 0.0 and whose alpha value is 1.0.

purpleColor

+ (NSColor *)**purpleColor**

Returns an NSColor in NSCalibratedRGBColorSpace whose RGB value is 0.5, 0.0, 0.5 and whose alpha value is 1.0.

redColor

+ (NSColor *)**redColor**

Returns an NSColor in NSCalibratedRGBColorSpace whose RGB value is 1.0, 0.0, 0.0 and whose alpha value is 1.0.

See also: – **redComponent**

scrollBarColor

+ (NSColor *)**scrollBarColor**

Returns the system color used for scroll “bars”—that is, for the groove in which a scroller’s knob moves. By default, this color is gray on the Macintosh, and COLOR_SCROLLBAR on Windows. On Windows, when a scroll bar is dragged, its color changes to the return value of **selectedControlColor**; on the Macintosh, however, its color does not change. For general information about system colors, see the “System Colors” section of the class description, above.

selectedControlColor

+ (NSColor *)**selectedControlColor**

Returns the system color used for the face of a selected control—that is a control being clicked or dragged. By default, this color is white on the Macintosh, and COLOR_HIGHLIGHT on Windows. For general information about system colors, see the “System Colors” section of the class description, above.

See also: + **selectedControlTextColor**

selectedControlTextColor

+ (NSColor *)**selectedControlTextColor**

Returns the system color used for text in a selected control—that is a control being clicked or dragged. By default, this color is black on the Macintosh, and COLOR_HIGHLIGHTTEXT on Windows. For general information about system colors, see the “System Colors” section of the class description, above.

See also: + **selectedControlColor**

selectedKnobColor

+ (NSColor *)**selectedKnobColor**

Returns the system color used for the slider knob when it is selected—that is, dragged. By default, this color is light blue on the Macintosh, and COLOR_HIGHLIGHT on Windows. For general information about system colors, see the “System Colors” section of the class description, above.

See also: **+ knobColor**

selectedMenuItemColor

+ (NSColor *)**selectedMenuItemColor**

Returns the system color used for the face of selected menu items. By default, this color is white on the Macintosh, and COLOR_HIGHLIGHT on Windows. For general information about system colors, see the "System Colors" section of the class description, above.

See also: **+ selectedMenuItemTextColor**

selectedMenuItemTextColor

+ (NSColor *)**selectedMenuItemTextColor**

Returns the system color used for the text in menu items. By default, this color is black on the Macintosh, and COLOR_HIGHLIGHTTEXT on Windows. For general information about system colors, see the "System Colors" section of the class description, above.

See also: **+ selectedMenuItemColor**

selectedTextBackgroundColor

+ (NSColor *)**selectedTextBackgroundColor**

Returns the system color used for the background of selected text. By default, this color is light gray on the Macintosh, and COLOR_HIGHLIGHT on Windows. For general information about system colors, see the "System Colors" section of the class description, above.

See also: + **selectedTextColor**

selectedTextColor

+ (NSColor *)**selectedTextColor**

Returns the system color used for selected text. By default, this color is black on the Macintosh, and COLOR_HIGHLIGHTTEXT on Windows. For general information about system colors, see the "System Colors" section of the class description, above.

See also: + **selectedTextBackgroundColor**

setIgnoresAlpha:

+ (void)**setIgnoresAlpha:**(BOOL)*flag*

If *flag* is YES, The application won't support alpha. In this case, no opacity slider is displayed in the color panel, and colors dragged in or pasted have their alpha values set to 1.0. Applications which need to import

alpha can invoke this method with *flag* set to NO and explicitly make colors opaque in cases where it matters to them.

See also: **+ ignoresAlpha**, **– alphaComponent**

shadowColor

+ (NSColor *)**shadowColor**

Returns the system color that represents the virtual shadows cast by raised objects on the screen. This method is invoked by **shadowWithLevel:**. By default, the color it returns is black on the Macintosh, and COLOR_3DDKSHADOW on Windows. For general information about system colors, see the "System Colors" section of the class description, above.

See also: **– shadowWithLevel:**

textBackgroundColor

+ (NSColor *)**textBackgroundColor**

Returns the system color used for the text background. By default, this color is white on the Macintosh, and COLOR_WINDOW on Windows. When text is selected, its background color changes to the return value of **selectedTextBackgroundColor**. For general information about system colors, see the "System Colors" section of the class description, above.

See also: + **textColor**

textColor

+ (NSColor *)**textColor**

Returns the system color used for text. By default, this color is black on the Macintosh, and COLOR_WINDOWTEXT on Windows. When text is selected, its background color changes to the return value of **selectedTextColor**. For general information about system colors, see the "System Colors" section of the class description, above.

See also: + **textBackgroundColor**

whiteColor

+ (NSColor *)**whiteColor**

Returns an NSColor in NSCalibratedWhiteColorSpace whose grayscale and alpha values are both 1.0.

See also: **– whiteComponent**

windowFrameColor

+ (NSColor *)**windowFrameColor**

Returns the system color used for window frames, except for their text. By default, this color is gray on the Macintosh, and COLOR_ACTIVEBORDER on Windows. For general information about system colors, see the “System Colors” section of the class description, above.

See also: **+ windowFrameTextColor**

windowFrameTextColor

+ (NSColor *)**windowFrameTextColor**

Returns the system color used for the text in window frames. By default, this color is black on the Macintosh, and COLOR_CAPTIONTEXT on Windows. For general information about system colors, see the “System Colors” section of the class description, above.

See also: **+ windowFrameColor**

yellowColor

+ (NSColor *)**yellowColor**

Returns an NSColor in NSCalibratedRGBColorSpace whose RGB value is 1.0, 1.0, 0.0 and whose alpha value is 1.0.

See also: **– yellowComponent**

Instance Methods

alphaComponent

– (float)**alphaComponent**

Returns the receiver’s alpha (opacity) component. Returns 1.0 (opaque) if the receiver has no alpha component.

See also: **– getCyan:magenta:yellow:black:alpha:**, **– getHue:saturation:brightness:alpha:**, **– getRed:green:blue:alpha:**, **– getWhite:alpha:**

blackComponent

– (float)**blackComponent**

Returns the receiver's black component. Raises an exception if the receiver isn't a CMYK color.

See also: **– getCyan:magenta:yellow:black:alpha:**

blendedColorWithFraction:ofColor:

– (NSColor *)**blendedColorWithFraction:**(float)*fraction*
ofColor:(NSColor *)*color*

Creates and returns an NSColor in NSCalibratedRGBColorSpace whose component values are a weighted sum of the receiver's and *color*'s. The method converts *color* and a copy of the receiver to RGB, and then sets each component of the returned color to *fraction* of *color*'s value plus 1 – *fraction* of the receiver's. Returns **nil** if the colors can't be converted to NSCalibratedRGBColorSpace.

blueComponent

– (float)**blueComponent**

Returns the receiver's blue component. Raises an exception if the receiver isn't an RGB color.

See also: **– getRed:green:blue:alpha:**

brightnessComponent

– (float)**brightnessComponent**

Returns the brightness component of the HSB color equivalent to the receiver. Raises an exception if the receiver isn't an RGB color.

See also: **– getHue:saturation:brightness:alpha:**

catalogNameComponent

– (NSString *)**catalogNameComponent**

Returns the name of the catalog containing the receiver's name, or raises an exception if the receiver's color space isn't NSNamedColorSpace.

See also: **+ colorWithCatalogName:colorName:**, **– colorNameComponent**,
– localizedCatalogNameComponent

colorNameComponent

– (NSString *)**colorNameComponent**

Returns the receiver's name, or raises an exception if the receiver's color space isn't NSNamedColorSpace.

See also: **+ colorWithCatalogName:colorName:**, **– catalogNameComponent**, **– localizedCatalogNameComponent**

colorSpaceName

– (NSString *)**colorSpaceName**

Returns the name of the receiver's color space. This method should be implemented in subclasses of NSColor.

See also: **– colorUsingColorSpaceName:**, **– colorUsingColorSpaceName:device:**

colorUsingColorSpaceName:

– (NSColor *)**colorUsingColorSpaceName:**(NSString *)*colorSpace*

Creates and returns an NSColor whose color is the same as the receiver's, except that the new NSColor is in the color space named *colorSpace*. If *colorSpace* is **nil**, the most appropriate color space is used.

Returns **nil** if the specified conversion cannot be done.

See also: **– colorSpaceName**

colorUsingColorSpaceName:device:

– (NSColor *)**colorUsingColorSpaceName:**(NSString *)*colorSpace*
device:(NSDictionary *)*deviceDescription*

Creates and returns an NSColor whose color is the same as the receiver's, except that the new NSColor is in the color space named *colorSpace* and is specific to the device described by *deviceDescription.* Device descriptions can be obtained from windows, screens, and printers with the **deviceDescription** method. If *colorSpace* is **nil**, the most appropriate color space is used.

If *deviceDescription* is **nil,** the current device (as obtained from the currently lockFocus'ed view's window or, if printing, the current printer) is used.

Returns **nil** if the specified conversion cannot be done.

See also: **– colorSpaceName**, **– colorUsingColorSpaceName:**

colorWithAlphaComponent:

– (NSColor *)**colorWithAlphaComponent:**(float)*alpha*

Creates and returns an NSColor that has the same color space and component values as the receiver, except that its alpha component is *alpha*. If the receiver's color space doesn't include an alpha component, the receiver is returned. A subclass which has explicit opacity components should override this method to return a color with the specified *alpha*.

See also: **– alphaComponent**, **– blendedColorWithFraction:ofColor:**

cyanComponent

– (float)**cyanComponent**

Returns the receiver's cyan component. Raises an exception if the receiver isn't a CMYK color.

See also: **– getCyan:magenta:yellow:black:alpha:**

drawSwatchInRect:

– (void)**drawSwatchInRect:**(NSRect)*rect*

Draws the current color in the rectangle *rect*. Subclasses adorn the rectangle in some manner to indicate the type of color. This method is invoked by color wells, swatches, and other user-interface objects that need to display colors.

getCyan:magenta:yellow:black:alpha:

– (void)**getCyan:**(float *)*cyan*
magenta:(float *)*magenta*
yellow:(float *)*yellow*
black:(float *)*black*
alpha:(float *)*alpha*

Returns the receiver's CMYK and alpha values in the respective arguments. If NULL is passed in as an argument, the method doesn't set that value. Raises an exception if the receiver isn't a CMYK color.

See also: **– alphaComponent**, **– blackComponent**, **– cyanComponent**, **– magentaComponent**, **– yellowComponent**

getHue:saturation:brightness:alpha:

– (void)**getHue:**(float *)*hue*
saturation:(float *)*saturation*
brightness:(float *)*brightness*
alpha:(float *)*alpha*

Returns the receiver's HSB and alpha values in the respective arguments. If NULL is passed in as an argument, the method doesn't set that value. Raises an exception if the receiver isn't an RGB color.

See also: **– alphaComponent**, **– brightnessComponent**, **– hueComponent**, **– saturationComponent**

getRed:green:blue:alpha:

– (void)**getRed:**(float *)*red*
green:(float *)*green*
blue:(float *)*blue*
alpha:(float *)*alpha*

Returns the receiver's RGB and alpha values in the respective arguments. If NULL is passed in as an argument, the method doesn't set that value. Raises an exception if the receiver isn't an RGB color.

See also: **– alphaComponent**, **– blueComponent**, **– greenComponent**, **– redComponent**

getWhite:alpha:

– (void)**getWhite:**(float *)*white*
alpha:(float *)*alpha*

Returns the receiver's grayscale and alpha values in the respective arguments. If NULL is passed in as an argument, the method doesn't set that value. Raises an exception if the receiver isn't a grayscale color.

See also: **– alphaComponent**, **– whiteComponent**

greenComponent

– (float)**greenComponent**

Returns the receiver's green component. Raises an exception if the receiver isn't an RGB color.

See also: **– getRed:green:blue:alpha:**

highlightWithLevel:

– (NSColor *)**highlightWithLevel:**(float)*highlightLevel*

Returns an NSColor in NSCalibratedRGBColorSpace that represents a blend between the receiver and the highlight color—that is, the NSColor returned by **highlightColor**. The highlight color's contribution to the blend depends on *highlightLevel*, which should be a number between 0.0 and 1.0. (A *highlightLevel* below 0.0 is interpreted as 0.0; a *highlightLevel* above 1.0 is interpreted as 1.0.)

Returns **nil** if the colors can't be converted to NSCalibratedRGBColorSpace. Invoke this method when you want to brighten the receiving NSColor for use in highlights.

See also: – **shadowWithLevel:**

hueComponent

– (float)**hueComponent**

Returns the hue component of the HSB color equivalent to the receiver. Raises an exception if the receiver isn't an RGB color.

See also: **– getHue:saturation:brightness:alpha:**

localizedCatalogNameComponent

– (NSString *)**localizedCatalogNameComponent**

Like **catalogNameComponent**, but returns a localized string. This string may be displayed in user-interface items like color pickers.

See also: **+ colorWithCatalogName:colorName:**, **– colorNameComponent**

localizedColorNameComponent

– (NSString *)**localizedColorNameComponent**

Like **colorNameComponent**, but returns a localized string. This string may be displayed in user-interface items like color pickers.

See also: **+ colorWithCatalogName:colorName:**, **– catalogNameComponent**, **– colorNameComponent**, **– localizedCatalogNameComponent**

magentaComponent

– (float)**magentaComponent**

Returns the receiver's magenta component. Raises an exception if the receiver isn't a CMYK color.

See also: **– getCyan:magenta:yellow:black:alpha:**

redComponent

– (float)**redComponent**

Returns the receiver's red component. Raises an exception if the receiver isn't an RGB color.

See also: **– getRed:green:blue:alpha:**

saturationComponent

– (float)**saturationComponent**

Returns the saturation component of the HSB color equivalent to the receiver. Raises an exception if the receiver isn't an RGB color.

See also: **– getHue:saturation:brightness:alpha:**

set

– (void)**set**

Sets the color of subsequent PostScript drawing to the color that the receiver represents. If the application is drawing to the screen rather than printing, this method also sets the current drawing context's alpha value to the value returned by **alphaComponent**; if the color doesn't know about alpha, it's set to 1.0. This method should be implemented in subclasses.

shadowWithLevel:

– (NSColor *)**shadowWithLevel:**(float)*shadowLevel*

Returns an NSColor in NSCalibratedRGBColorSpace that represents a blend between the receiver and the shadow color—that is, the NSColor returned by **shadowColor**. The shadow color's contribution to the blend depends on *shadowLevel*, which should be a number between 0.0 and 1.0. (A *shadowLevel* below 0.0 is interpreted as 0.0; a *shadowLevel* above 1.0 is interpreted as 1.0.)

Returns **nil** if the colors can't be converted to NSCalibratedRGBColorSpace. Invoke this method when you want to darken the receiving NSColor for use in shadows.

See also: – **highlightWithLevel:**

whiteComponent

– (float)**whiteComponent**

Returns the receiver's white component. Raises an exception if the receiver isn't a grayscale color.

See also: **– getWhite:alpha:**

writeToPasteboard:

– (void)**writeToPasteboard:**(NSPasteboard *)*pasteBoard*

Writes the receiver's data to the pasteboard, unless the pasteboard doesn't support color data (in which case the method does nothing).

See also: **+ colorFromPasteboard:**

yellowComponent

– (float)**yellowComponent**

Returns the receiver's yellow component. Raises an exception if the receiver isn't a CMYK color.

See also: **– getCyan:magenta:yellow:black:alpha:**

Notifications

NSSystemColorsDidChangeNotification

Sent when the system colors have been changed (such as through a system control panel interface).

This notification contains no notification object and no userInfo dictionary.

NSColorList

Inherits From:	NSObject
Conforms To:	NSCoding NSObject (NSObject)
Declared In:	AppKit/NSColorList.h

Class Description

An NSColorList is an ordered list of NSColors, identified by keys. Instances of NSColorList, or more simply, *color lists*, are used to manage named lists of NSColors. NSColorPanel's list-mode color picker uses instances of NSColorList to represent any lists of colors that come with the system, as well as any lists created by the user. An application can use NSColorList to manage document-specific color lists, which may be added to an application's NSColorPanel using its **attachColorList:** method.

An NSColorList is similar to a dictionary object: An NSColor is added to, looked up in, and removed from the list by specifying its key, which is an NSString. These keys are used to identify the colors in the list and are used to display the color to the user in the color panel. In addition, colors can be inserted at specified positions in the list.

The color list has a name, specified when you create the object using either the **initWithName:** or **initWithName:fromFile:** method.

Instances of NSColorList are created for all user-created color lists (those in the color panel) and various color catalogs available on the system.

An NSColorList saves and retrieves its colors from files with the extension "**.clr**" in directories defined by a standard search path. To access all the color lists in the standard search path, use the **availableColorLists** method; this returns an array of NSColorLists, from which you can retrieve the individual color lists by name.

The standard search path for color lists is:

- **/NextLibrary/Colors**
- **/LocalLibrary/Colors**
- **~/Library/Colors**

NSColorList reads color list files in several different formats; it saves color lists using the archiver API.

NSColorList posts an NSColorListChanged notification when a color list is changed.

Adopted Protocols

NSCoding

– encodeWithCoder:
– initWithCoder:

Method Types

Initializing an NSColorList

– initWithName:
– initWithName:fromFile:

Getting All Color Lists

+ availableColorLists

Getting a Color List by Name

+ colorListNamed:
– name

Managing Colors by Key

– allKeys
– colorWithKey:
– insertColor:key:atIndex:
– removeColorWithKey:
– setColor:forKey:

Editing

– isEditable

Writing and Removing Files

– removeFile
– writeToFile:

Class Methods

availableColorLists

+ (NSArray *)**availableColorLists**

Returns an array of all NSColorLists found in the standard color list directories. Color lists created at run time aren't included in this list unless they're saved into one of the standard color list directories.

See also: **+ colorListNamed:**

colorListNamed:

+ (NSColorList *)**colorListNamed:**(NSString *)*name*

Searches the array that's returned by **availableColorLists** and returns the NSColorList named *name*, or **nil** if no such color list exists. *name* must not include the "**.clr**" suffix.

See also: **– name**

Instance Methods

allKeys

– (NSArray *)**allKeys**

Returns an array of NSString objects that contains all the keys by which the NSColors are stored in the NSColorList. The length of this array equals the number of colors, and its contents are arranged according to the ordering specified when the colors were inserted.

colorWithKey:

– (NSColor *)**colorWithKey:**(NSString *)*key*

Returns the NSColor associated with *key*, or **nil** if there is none.

initWithName:

– (id)**initWithName:**(NSString *)*name*

Initializes and returns the receiver, registering it under the specified *name* if *name* isn't in use already. This method invokes **initWithName:fromFile:** with a **fromFile:** argument of **nil**, indicating that the color list doesn't need to be initialized from a file.

initWithName:fromFile:

– (id)**initWithName:**(NSString *)*name*
fromFile:(NSString *)*path*

Initializes and returns the receiver, registering it under the specified *name* if *name* isn't in use already. *path* should be the full path to the file for the color list; *name* should be the name of the file for the color list (minus the "**.clr**" extension). A **nil** *path* indicates that the color list should be initialized with no colors.

insertColor:key:atIndex:

– (void)**insertColor:**(NSColor *)*color*
key:(NSString *)*key*
atIndex:(unsigned)*location*

Inserts *color* at the specified location in the color list (which is numbered starting with 0). If the list already contains a color with the same key at a different location, it's removed from the old location. This method posts the NSColorListChangedNotification notification to the default notification center. It raises the NSColorListNotEditableException exception if the color list isn't editable.

See also: **– colorWithKey**:, **– removeColorWithKey:**, **– setColor:forKey:**

isEditable

– (BOOL)**isEditable**

Returns YES if the color list can be modified. This depends on the source of the list: If it came from a write-protected file, this method returns NO.

name

– (NSString *)**name**

Returns the name of the NSColorList.

removeColorWithKey:

– (void)**removeColorWithKey:**(NSString *)*key*

Removes the color associated with *key* from the list. This method does nothing if the list doesn't contain the key. This method posts the NSColorListChangedNotification notification to the default notification center. It raises the NSColorListNotEditableException exception if the color list is not editable.

See also: **– insertColor:key:atIndex:**, **– setColor:forKey:**

removeFile

– (void)**removeFile**

Removes the file from which the list was created, if the file is in a standard search path and is owned by the user. The receiver is removed from the list of available color lists returned by **availableColorLists**, but isn't released.

setColor:forKey:

– (void)**setColor:**(NSColor *)*color*
forKey:(NSString *)*key*

Associates the specified NSColor with *key*. If the list already contains *key*, this method sets the corresponding color to *color*; otherwise, it inserts *color* at the end of the list by invoking **insertColor:key:atIndex:**.

See also: **– colorWithKey:, – insertColor:key:atIndex:**, **– removeColorWithKey:**

writeToFile:

– (BOOL)**writeToFile:**(NSString *)*path*

If *path* is a directory, saves the NSColorList in a file named *listname***.clr** in that directory (where *listname* is the name with which the NSColorList was initialized). If *path* includes a file name, this method saves the file under that name. If *path* is **nil**, this method saves the file as *listname***.clr** in the standard location. Returns YES upon success and NO if it fails to write the file.

See also: **– removeFile**

Notifications

NSColorListChangedNotification

This notification contains a notification object but no userInfo dictionary. The notification object is the NSColorList object that changed.

This notification is posted whenever a color list changes.

NSColorPanel

Inherits From: NSPanel : NSWindow : NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSColorPanel.h

Class Description

NSColorPanel provides a standard user interface for selecting color in an application. It provides a number of standard color selection modes, and, with the NSColorPickingDefault and NSColorPickingCustom protocols, allows an application to add its own color selection modes. It allows the user to save swatches containing frequently used colors. Once set, these swatches are displayed by NSColorPanel in any application where it is used, giving the user color consistency between applications. NSColorPanel enables users to capture a color anywhere on the screen for use in the active application, or to drag a color from the color panel into an application view.

When you press the color panel's "Set" button, NSColorPanel sends a **changeColor:** message to the first responder. It also sends its action message (set by **setAction:**) to its target object (set by **setTarget:**), provided that neither the action nor the target is **nil**. NSColorPanel also sends its action to its target whenever you select a color in the color panel.

An application has only one instance of NSColorPanel, the shared instance. Invoking the **sharedColorPanel:** method returns the shared instance of NSColorPanel, instantiating it if necessary.

You can put NSColorPanel in any application created with Interface Builder by adding the "Colors..." item from the Menu palette to the application's menu.

Color Mask and Color Modes

The color mask determines which of the color modes are enabled for NSColorPanel. This mask is set before you initialize a new instance of NSColorPanel. NSColorPanelAllModesMask represents the logical OR of the other color mask constants: It causes the NSColorPanel to display all standard color pickers. When initializing a new instance of NSColorPanel, you can logically OR any combination of color mask constants to restrict the available color modes.

Mode	Color Mask Constant
Grayscale-Alpha	NSColorPanelGrayModeMask

Mode	Color Mask Constant
Red-Green-Blue	NSColorPanelRGBModeMask
Cyan-Yellow-Magenta-Black	NSColorPanelCMYKModeMask
Hue-Saturation-Brightness	NSColorPanelHSBModeMask
Custom palette	NSColorPanelCustomPaletteModeMask
Custom color list	NSColorPanelColorListModeMask
Color wheel	NSColorPanelWheelModeMask
All of the above	NSColorPanelAllModesMask

The NSColorPanel's color mode mask is set using the class method **setPickerMask:**. The mask must be set before creating an application's instance of NSColorPanel.

When an application's instance of NSColorPanel is masked for more than one color mode, your program can set its active mode by invoking the **setMode:** method with a color mode constant as its argument; the user can set the mode by clicking buttons on the panel. Here are the standard color modes and mode constants:

Mode	Color Mode Constant
Grayscale-Alpha	NSGrayModeColorPanel
Red-Green-Blue	NSRGBModeColorPanel
Cyan-Yellow-Magenta-Black	NSCMYKModeColorPanel
Hue-Saturation-Brightness	NSHSBModeColorPanel
Custom palette	NSCustomPaletteModeColorPanel
Custom color list	NSColorListModeColorPanel
Color wheel	NSWheelModeColorPanel

In grayscale-alpha, red-green-blue, cyan-magenta-yellow-black, and hue-saturation-brightness modes, the user adjusts colors by manipulating sliders. In the custom palette mode, the user can load an NSImage file (TIFF or EPS) into the NSColorPanel, then select colors from the image. In custom color list mode, the user can create and load lists of named colors. The two custom modes provide NSPopUpButtons for loading and saving files. Finally, color wheel mode provides a simplified control for selecting colors.

If a color panel has been used, it uses whatever mode it was in last as the default mode when NSColorPanelAllModesMask is used to initialize the NSColorPanel. Otherwise, it uses color wheel mode.

Associated Classes and Protocols

The NSColorList class provides an API for managing custom color lists. The NSColorPanel methods **attachColorList:** and **detachColorList:** let your application add and remove custom lists from the NSColorPanel's user interface.

The protocols NSColorPickingDefault and NSColorPickingCustom provide an API for adding custom color selection to the user interface. The NSColorPicker class implements the NSColorPickingDefault protocol; you can subclass NSColorPicker and implement the NSColorPickingCustom protocol in your subclass to create your own user interface for color selection. NSColorPanel dynamically loads NSColorPickers from the following directories: ~/Library/ColorPickers/, /LocalLibrary/ColorPickers/, and /NextLibrary/ColorPickers/.

Method Types

Creating the NSColorPanel

+ sharedColorPanel
+ sharedColorPanelExists

Setting color picker modes

+ setPickerMask:
+ setPickerMode:

Setting the NSColorPanel

– accessoryView
– isContinuous
– mode
– setAccessoryView:
– setAction:
– setContinuous:
– setMode:
– setShowsAlpha:
– setTarget:
– showsAlpha

Attaching a color list

– attachColorList:
– detachColorList:

Setting color

+ dragColor:withEvent:fromView:
– setColor:

Getting color information

– alpha
– color

Class Methods

dragColor:withEvent:fromView:

+ (BOOL)**dragColor:**(NSColor *)*color*
withEvent:(NSEvent *)*anEvent*
fromView:(NSView *)*sourceView*

Drags *color* into a destination view from *sourceView*. This method is usually invoked by the **mouseDown:** method of *sourceView*. The dragging mechanism handles all subsequent events.

Because it is a class method, **dragColor:withEvent:fromView:** can be invoked whether or not the instance of NSColorPanel exists. Returns YES.

setPickerMask:

+ (void)**setPickerMask:**(int)*mask*

Accepts as a parameter one or more logically ORed color mode masks (defined in the header file **AppKit/NSColorPanel.h**):

- NSColorPanelGrayModeMask
- NSColorPanelRGBModeMask
- NSColorPanelCMYKModeMask
- NSColorPanelHSBModeMask
- NSColorPanelCustomPaletteModeMask
- NSColorPanelColorListModeMask
- NSColorPanelWheelModeMask
- NSColorPanelAllModesMask

This determines which color selection modes will be available in an application's NSColorPanel. This method only has an effect before NSColorPanel is instantiated.

If you create a class that implements the color picking protocols (NSColorPickingDefault and NSColorPickingCustom), you may want to give it a unique mask—one different from those defined for the standard color pickers. To display your color picker, your application will need to logically OR that unique mask with the standard color mask constants when invoking this method.

See also: **+ setPickerMode:**

setPickerMode:

+ (void)**setPickerMode:**(int)*mode*

Sets the color panel's initial picker to *mode*, which may be one of the symbolic constants described in the class description (declared in the header file **AppKit/NSColorPanel.h**). The mode determines which picker will initially be visible. This method may be called at any time, whether or not an application's NSColorPanel has been instantiated.

See also: **+ setPickerMask:**, **– setMode:**

sharedColorPanel

+ (NSColorPanel *)**sharedColorPanel**

Creates if necessary and returns the shared NSColorPanel.

sharedColorPanelExists

+ (BOOL)**sharedColorPanelExists**

Returns YES if the NSColorPanel has been created already.

See also: **+ sharedColorPanel**

Instance Methods

accessoryView

– (NSView *)**accessoryView**

Returns the accessory view, or **nil** if there is none.

See also: **– setAccessoryView:**

alpha

– (float)**alpha**

Returns the NSColorPanel's current alpha value based on its opacity slider. Returns 1.0 (opaque) if the panel has no opacity slider.

See also: **– setShowsAlpha:**, **– showsAlpha**

attachColorList:

– (void)**attachColorList:**(NSColorList *)*colorList*

Adds the specified list of NSColors to all the color pickers in the color panel that display color lists by invoking **attachColorList:** on all color pickers in the application.

An application should use this method to add an NSColorList saved with a document in its file package or in a directory other than NSColorList's standard search directories.

See also: **– detachColorList:**

color

– (NSColor *)**color**

Returns the currently selected color in the NSColorPanel.

See also: **– setColor:**

detachColorList:

– (void)**detachColorList:**(NSColorList *)*colorList*

Removes the specified list of NSColors from all the color pickers in the color panel that display color lists by invoking **detachColorList:** on all color pickers in the application.

Your application should use this method to remove an NSColorList saved with a document in its file package or in a directory other than NSColorList's standard search directories.

See also: **– attachColorList:**

isContinuous

– (BOOL)**isContinuous**

Returns whether or not the NSColorPanel continuously sends the action message to the target as the user manipulates the color picker.

See also: **– setContinuous:**

mode

– (int)**mode**

Returns the color picker mode of the NSColorPanel. The mode constants for the standard color pickers are listed in the class description.

See also: **+ setPickerMode:**, **– setMode:**

setAccessoryView:

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

Sets the accessory view displayed in the NSColorPanel to *aView*. The accessory view can be any custom view that you want to display with NSColorPanel, such as a view offering color blends in a drawing program. The accessory view is displayed below the color picker and above the color swatches in the NSColorPanel. The NSColorPanel automatically resizes to accommodate the accessory view. Returns the previous accessory view, if there was one; otherwise, returns **nil**.

See also: **– accessoryView**

setAction:

– (void)**setAction:**(SEL)*action*

Sets the action message to *action*. When you select a color in the color panel, or press the “Set” button, NSColorPanel sends its action to its target, provided that neither the action nor the target is **nil**. The action is **nil** by default.

See also: **– setTarget:**

setColor:

– (void)**setColor:**(NSColor *)*color*

Sets the color of the NSColorPanel to *color*. This method posts the NSColorPanelChangedNotification notification with the receiving object to the default notification center.

See also: **– color**

setContinuous:

– (void)**setContinuous:**(BOOL)*flag*

Sets the NSColorPanel to send the action message to its target continuously as the color of the NSColorPanel is set by the user. Send this message with flag YES if, for example, you want to continuously update the color of the target.

See also: **– isContinuous**

setMode:

– (void)**setMode:**(int)*mode*

Sets the mode of the NSColorPanel if *mode* is one of the modes allowed by the color mask. The color mask is set when you first create the shared instance of NSColorPanel for an application. *mode* may be one of these symbolic constants described in the class description (and declared in the header file **AppKit/NSColorPanel.h**):

- NSGrayModeColorPanel
- NSRGBModeColorPanel
- NSCMYKModeColorPanel
- NSHSBModeColorPanel
- NSCustomPaletteModeColorPanel
- NSColorListModeColorPanel
- NSWheelModeColorPanel

See also: **+ setPickerMode:**, **– mode**

setShowsAlpha:

– (void)**setShowsAlpha:**(BOOL)*flag*

Tells the NSColorPanel whether or not to show alpha values and an opacity slider.

See also: **– alpha**, **– showsAlpha**

setTarget:

– (void)**setTarget:**(id)*target*

Sets the target of the NSColorPanel to *target*. When you select a color in the color panel, or press the “Set” button, NSColorPanel sends its action to its target, provided that neither the action nor the target is **nil**. The target is **nil** by default.

See also: **– setAction:**, **– setContinuous:**

showsAlpha

– (BOOL)**showsAlpha**

Returns whether or not the NSColorPanel shows alpha values and an opacity slider.

See also: **– alpha**, **– setShowsAlpha:**

Notifications

NSColorPanelColorChangedNotification

This notification contains a notification object but no userInfo dictionary. The notification object is the notifying NSColorPanel. This notification is posted when the NSColorPanel's color is set, as when **setColor:** is invoked.

Methods Implemented by Responders

changeColor:

– (void)**changeColor:**(id)*sender*

When the user presses the "Set" button of an NSColorPanel, the NSColorPanel sends a **changeColor:** action message to the first responder. You can override this method in any responder that needs to respond to a color change. *sender* is the id of the color panel.

NSColorPicker

Inherits From: NSObject

Conforms To: NSColorPickingDefault
NSObject (NSObject)

Declared In: AppKit/NSColorPicker.h

Class Description

NSColorPicker is an abstract superclass that implements the NSColorPickingDefault protocol. The NSColorPickingDefault and NSColorPickingCustom protocols define a way to add color pickers (custom user interfaces for color selection) to the NSColorPanel. The simplest way to implement a color picker is to create a subclass of NSColorPicker, instead of implementing the NSColorPickingDefault protocol in another kind of object. (To add functionality, implement the NSColorPickingCustom methods in your subclass.)

The NSColorPickingDefault protocol specification describes the details of implementing a color picker and adding it to your application's NSColorPanel; you should look there first for an overview of how NSColorPicker works. This specification is provided to document the specific behavior of NSColorPicker's methods.

Adopted Protocols

NSColorPickingDefault

– alphaControlAddedOrRemoved:
– attachColorList:
– detachColorList:
– initWithPickerMask:colorPanel:
– insertNewButtonImage:in:
– provideNewButtonImage
– setMode:
– viewSizeChanged:

Method Types

Initializing an NSColorPicker

– initWithPickerMask:colorPanel:

Getting the color panel

– colorPanel

Adding button images

– insertNewButtonImage:in:
– provideNewButtonImage

Setting the mode

– setMode:

Using color lists

– attachColorList:
– detachColorList:

Responding to a resized view

– viewSizeChanged:

Instance Methods

attachColorList:

– (void)**attachColorList:**(NSColorList *)*colorList*

Does nothing. Override to attach a color list to a color picker.

See also: **– detachColorList:**

colorPanel

– (NSColorPanel *)**colorPanel**

Returns the NSColorPanel that owns this NSColorPicker.

detachColorList:

– (void)**detachColorList:**(NSColorList *)*colorList*

Does nothing. Override to detach a color list from a color picker.

See also: **– attachColorList:**

initWithPickerMask:colorPanel:

– (id)**initWithPickerMask:**(int)*mask*
colorPanel:(NSColorPanel *)*owningColorPanel*

Sets the color picker's color panel to *owningColorPanel*, caching the *owningColorPanel* value so it can later be returned by the **colorPanel** method. Returns self. Override this method to respond to the values in *mask* or do other custom initialization. If you override this method in a subclass, you should forward the message to **super** as part of the implementation.

See also: **– colorPanel**

insertNewButtonImage:in:

– (void)**insertNewButtonImage:**(NSImage *)*newButtonImage*
in:(NSButtonCell *)*buttonCell*

Sets *newButtonImage* as *buttonCell*'s image by invoking NSButtonCell's **setImage:** method. Called by the color panel to insert a new image into the specified cell. Override this method to customize *newButtonImage* before insertion in *buttonCell*.

See also: **– provideNewButtonImage**

provideNewButtonImage

– (NSImage *)**provideNewButtonImage**

Returns the button image for the color picker. The color panel will place this image in the mode button that the user uses to select this picker. (This is the same image that the color panel uses as an argument when sending the **insertNewButtonImage:in:** message.) The default implementation looks in the color picker's bundle for a TIFF file named after the color picker's class, with the extension "**.tiff**".

See also: **– insertNewButtonImage:in:**

setMode:

– (void)**setMode:**(int)*mode*

Does nothing. Override to set the color picker's mode. Here are the standard color picking modes and mode constants (defined in **AppKit/NSColorPanel.h**):

Mode	Color Mode Constant
Grayscale-Alpha	NSGrayModeColorPanel

Mode	Color Mode Constant
Red-Green-Blue	NSRGBModeColorPanel
Cyan-Yellow-Magenta-Black	NSCMYKModeColorPanel
Hue-Saturation-Brightness	NSHSBModeColorPanel
Custom palette	NSCustomPaletteModeColorPanel
Custom color list	NSColorListModeColorPanel
Color wheel	NSWheelModeColorPanel

In grayscale-alpha, red-green-blue, cyan-magenta-yellow-black, and hue-saturation-brightness modes, the user adjusts colors by manipulating sliders. In the custom palette mode, the user can load an NSImage file (TIFF or EPS) into the NSColorPanel, then select colors from the image. In custom color list mode, the user can create and load lists of named colors. The two custom modes provide NSPopUpLists for loading and saving files. Finally, color wheel mode provides a simplified control for selecting colors.

viewSizeChanged:

– (void)**viewSizeChanged:**(id)*sender*

Does nothing. Override to respond to a size change.

NSColorWell

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

Class Description

NSColorWell is an NSControl for selecting and displaying a single color value. An example of an NSColorWell object (or simply *color well*) is found in NSColorPanel, which uses a color well to display the current color selection. A color well is available from the Palettes panel of Interface Builder.

An application can have one or more active color wells. You can activate multiple color wells by invoking the **activate:** method with NO as its argument. When a mouse-down event occurs on a color well's border, it becomes the only active color well. When a color well becomes active, it brings up the color panel also.

The **mouseDown:** method enables a color well to send its color to another color well or any other subclass of NSView that implements the NSDraggingDestination protocol.

Method Types

Drawing

– drawWellInside:

Activating

– activate:
– deactivate
– isActive

Managing Color

– color
– setColor:
– takeColorFrom:

Managing Borders

– isBordered
– setBordered:

Instance Methods

activate:

– (void)**activate:**(BOOL)*exclusive*

Activates the NSColorWell, displays the Color panel, and makes the NSColorPanel's current color the same as its own. If *exclusive* is YES, deactivates any other color wells; if NO, keeps them active. Redraws the receiver. An active color well will have its color updated when the NSColorPanel's current color changes. Any color well that shows its border highlights the border when it's active.

See also: **– deactivate**, **– isActive**

color

– (NSColor *)**color**

Returns the color of the NSColorWell.

See also: **– setColor:**, **– takeColorFrom:**

deactivate

– (void)**deactivate**

Deactivates the NSColorWell and redraws it.

See also: **– activate:**, **– isActive**

drawWellInside:

– (void)**drawWellInside:**(NSRect)*insideRect*

Draws the colored area inside the NSColorWell at the location specified by *insideRect* without drawing borders.

isActive

– (BOOL)**isActive**

Indicates whether the NSColorWell is active.

isBordered

– (BOOL)**isBordered**

Indicates whether the NSColorWell is bordered.

See also: **– setBordered:**

setBordered:

– (void)**setBordered:**(BOOL)*bordered*

Places or removes a border on the NSColorWell, depending on *bordered*, and redraws the receiver.

See also: **– isBordered**

setColor:

– (void)**setColor:**(NSColor *)*color*

Sets the color of the NSColorWell to *color* and redraws the receiver.

See also: **– color**, **– takeColorFrom:**

takeColorFrom:

– (void)**takeColorFrom:**(id)*sender*

Changes the color of the NSColorWell to that of *sender*.

See also: **– color**, **– setColor:**

NSComboBox

Inherits From: NSTextField : NSControl : NSView : NSResponder : NSObject

Conforms To: NSObject (NSObject)
NSCoding
NSCopying

Declared In: AppKit/NSComboBox.h

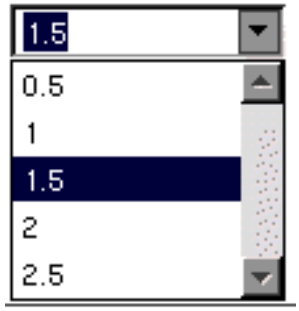
Class Description

An NSComboBox is a kind of NSControl that allows you to either enter text directly (as you would with an NSTextField), or click the attached arrow at the right of the combo box and select from a displayed ("pop-up") list of items. Use this control whenever you want the user to enter information that can be selected from a finite list of options. Note that while you can construct your NSComboBox so that users are restricted to only selecting items from the combo box's pop-up list, this isn't the combo box's normal behavior: a user can either select an item from the list, or enter text that may or may not be contained in the pop-up list.

While the pop-up list is visible, typing into the text field causes an incremental search to be performed on the list. If there's a match, the selection in the pop-up list changes to reflect the match.

The NSComboBox normally looks like this:

When you click the downward-pointing arrow at the right-hand side of the text field the pop-up list appears, like this:



If there isn't sufficient room for the pop-up list to be displayed below the text field, it's instead displayed above the text field. Selecting an item from the list, clicking anywhere outside the control, or activating another window dismisses the pop-up list.

Providing Data for the Combo Box's Pop-Up List

The NSComboBox control can be set up to populate the pop-up list either from an internal item list or from an object that you provide, called its *data source*. If you use a data source, your data source object can store items in any way, but it must be able to identify them by an integer index. See the NSComboBoxDataSource informal protocol specification for more information on constructing an NSComboBox data source.

NSComboBox provides a complete set of methods that allow you to add, insert, and delete items in the internal item list for combo boxes that don't use a data source.

Use **setUsesDataSource:** to specify whether a given combo box uses a data source or maintains an internal list of items. A combo box can only use one or the other; for instance, if you construct combo box that uses a data source and then attempt to execute an item-oriented method—such as **addItemWithObjectValue:**—a warning will be logged and the method will have no effect.

Interacting with the Text Field

Because NSComboBox is a type of NSControl, you typically use the methods provided by the NSControl class—such as **stringValue**, **floatValue**, or **intValue**—when working with the contents of the combo box's text field; see the NSControl class specification for more information on these methods. NSControl's **set...Value:** methods are also useful, primarily when initializing a combo box. For instance, the following excerpt shows how to "pre-select" the third item in the list of a combo box that maintains an internal item list:

```
[myComboBox selectItemAtIndex:2];// List items start at index 0
[myComboBox setObjectValue:[myComboBox objectValueOfSelectedItem]];
```

To do the same thing for a combo box that relies upon a data source, use:

```
[myComboBox selectItemAtIndex:2];
[myComboBox setObjectValue:[myComboBoxDataSource comboBox:myComboBox
    objectValueForItemAtIndex:[myComboBox indexOfSelectedItem]]];
```

Note that NSComboBox is a also a subclass of NSTextField, and thus inherits all of NSTextField's methods. NSComboBox relies heavily upon its cell class, NSComboBoxCell. NSComboBoxCell is a NSTextFieldCell subclass, which combines a text field cell with a button cell.

Method Types

Setting display attributes

– hasVerticalScroller
– intercellSpacing
– itemHeight
– numberOfVisibleItems
– setHasVerticalScroller:
– setIntercellSpacing:
– setItemHeight:
– setNumberOfVisibleItems:

Setting a data source

– dataSource
– setDataSource:
– setUsesDataSource:
– usesDataSource

Working with an internal list

– addItemsWithObjectValues:
– addItemWithObjectValue:
– insertItemWithObjectValue:atIndex:
– objectValues
– removeAllItems
– removeItemAtIndex:
– removeItemWithObjectValue:
– numberOfItems

Manipulating the displayed list

– indexOfItemWithObjectValue:
– itemObjectValueAtIndex:
– noteNumberOfItemsChanged
– reloadData
– scrollItemAtIndexToTop:
– scrollItemAtIndexToVisible:

Manipulating the selection

– deselectItemAtIndex:
– indexOfSelectedItem
– objectValueOfSelectedItem
– selectItemAtIndex:
– selectItemWithObjectValue:

Encoding a ComboBox

– encodeWithCoder:
– initWithCoder:

Instance Methods

addItemsWithObjectValues:

– (void)**addItemsWithObjectValues:**(id)*objects*

Adds multiple objects to the end of the combo box's internal item list. This method logs a warning if **usesDataSource** returns YES.

addItemWithObjectValue:

– (void)**addItemWithObjectValue:**(id)*anObject*

Adds *anObject* to the end of the combo box's internal item list. This method logs a warning if **usesDataSource** returns YES.

dataSource

– (id)**dataSource**

Returns the object that provides the data displayed in the receiver's pop-up list. This method logs a warning if **usesDataSource** returns NO. See the class description and the NSComboBoxDataSource informal protocol specification for more information on combo box data source objects.

deselectItemAtIndex:

– (void)**deselectItemAtIndex:**(int)*index*

Deselects the pop-up list item at *index* if it's selected. If the selection does in fact change, this method posts an NSComboBoxSelectionDidChangeNotification to the default notification center.

See also: – **indexOfSelectedItem**, – **numberOfItems**, – **selectItemAtIndex:**

encodeWithCoder:

– (void)**encodeWithCoder:**(NSCoder *)*encoder*

Encodes the receiver using *encoder*. If the receiver uses a data source, the data source is conditionally encoded as well.

See also: – **initWithCoder:**

hasVerticalScroller

– (BOOL)**hasVerticalScroller**

Returns YES if the receiver will display a vertical scroller. Note that the scoller will be displayed even if the pop-up list contains fewer items than will fit in the area specified for display. Returns NO if the receiver won't display a vertical scroller.

See also: – **numberOfItems**, – **numberOfVisibleItems**

indexOfItemWithObjectValue:

– (int)**indexOfItemWithObjectValue:**(id)*anObject*

Searches the receiver's internal item list for *anObject* and returns the lowest index whose corresponding value is equal to *anObject*. Objects are considered equal if they have the same **id** or if **isEqual:** returns YES. If none of the objects in the receiver's internal item list are equal to *anObject*, **indexOfItemObjectValue:** returns NSNotFound. This method logs a warning if **usesDataSource** returns YES.

See also: – **selectItemWithObjectValue:**

indexOfSelectedItem

– (int)**indexOfSelectedItem**

Returns the index of the last item selected from the receiver's pop-up list, or -1 if no item is selected. Note that nothing is initially selected in a newly-initialized combo box.

See also: – **objectValueOfSelectedItem**

initWithCoder:

– (id)**initWithCoder:**(NSCoder *)*decoder*

Initializes a newly-allocated instance from data in *decoder*. If the decoded instance uses a data source, **initWithCoder:** decodes the data source as well. Returns **self**.

See also: – **encodeWithCoder:**

insertItemWithObjectValue:atIndex:

– (void)**insertItemWithObjectValue:**(id)*anObject* **atIndex:**(int)*index*

Inserts *anObject* at *index* in the combo box's internal item list, shifting the previous item at *index*—along with all following items—down one slot to make room. This method logs a warning if **usesDataSource** returns YES.

See also: – **addItemWithObjectValue:**, – **numberOfItems**

intercellSpacing

– (NSSize)**intercellSpacing**

Returns the horizontal and vertical spacing between cells in the receiver's pop-up list. The default spacing is (3.0, 2.0).

See also: – **itemHeight**, – **numberOfVisibleItems**

itemHeight

– (float)**itemHeight**

Returns the height of each item in the receiver's pop-up list. The default item height is 16.0.

See also: – **intercellSpacing**, – **numberOfVisibleItems**

itemObjectValueAtIndex:

– (id)**itemObjectValueAtIndex:**(int)*index*

Returns the object located at *index* within the receiver's internal item list. If *index* is beyond the end of the list, an NSRangeException is raised. This method logs a warning if **usesDataSource** returns YES.

See also: – **objectValueOfSelectedItem**

noteNumberOfItemsChanged

– (void)**noteNumberOfItemsChanged**

Informs the receiver that the number of items in its data source has changed, allowing the receiver to update the scrollers in its displayed pop-up list without actually reloading data into the receiver. This method is particularly useful for a data source that continually receives data in the background over a period of time, in which case the NSComboBox can remain responsive to the user while the data is received.

See the NSComboBoxDataSource informal protocol specification for information on the messages an NSComboBox sends to its data source.

See also: – **reloadData**

numberOfItems

– (int)**numberOfItems**

Returns the total number of items in the pop-up list.

See also: – **numberOfItemsInComboBox:** (NSComboBoxDataSource protocol),
– **numberOfVisibleItems**

numberOfVisibleItems

– (int)**numberOfVisibleItems**

Returns the maximum number of items visible at any one time in the pop-up list.

See also: – **numberOfItems**

objectValueOfSelectedItem

– (id)**objectValueOfSelectedItem**

Returns the object from the receiver's internal item list corresponding to the last item selected from the pop-up list, or **nil** if no item is selected. Note that nothing is initially selected in a newly-initialized combo box. This method logs a warning if **usesDataSource** returns YES.

See also: – **comboBox:objectValueForItemAtIndex:** (NSComboBoxDataSource protocol),
– **indexOfSelectedItem**

objectValues

– (NSArray *)**objectValues**

Returns as an array the receiver's internal item list. This method logs a warning if **usesDataSource** returns YES.

reloadData

– (void)**reloadData**

Marks the receiver as needing redisplay, so that it will reload the data for visible pop-up items and draw the new values.

See also: – **noteNumberOfItemsChanged**

removeAllItems

– (void)**removeAllItems**

Removes all items from the receiver's internal item list. This method logs a warning if **usesDataSource** returns YES.

See also: – **objectValues**

removeItemAtIndex:

– (void)**removeItemAtIndex:**(int)*index*

Removes the object at *index* from the receiver's internal item list and moves all items beyond *index* up one slot to fill the gap. The removed object receives a release message. This method raises an NSRangeException if *index* is beyond the end of the list, and logs a warning if **usesDataSource** returns YES.

removeItemWithObjectValue:

– (void)**removeItemWithObjectValue:**(id)*anObject*

Removes all occurrences of *anObject* from the receiver's internal item list. Objects are considered equal if they have the same **id** or if **isEqual:** returns YES. This method logs a warning if **usesDataSource** returns YES.

See also: – **indexOfItemWithObjectValue:**

scrollItemAtIndexToTop:

– (void)**scrollItemAtIndexToTop:**(int)*index*

Scrolls the receiver's pop-up list vertically so that the item specified by *index* is as close to the top as possible. The pop-up list need not be displayed at the time this method is invoked.

scrollItemAtIndexToVisible:

– (void)**scrollItemAtIndexToVisible:**(int)*index*

Scrolls the receiver's pop-up list vertically so that the item specified by *index* is visible. The pop-up list need not be displayed at the time this method is invoked.

selectItemAtIndex:

– (void)**selectItemAtIndex:**(int)*index*

Selects the pop-up list row at *index*. Posts NSComboBoxSelectionDidChangeNotification to the default notification center if the selection does in fact change. Note that this method does not alter the contents of the combo box's text field—see "Interacting with the Text Field" in the class description for more information.

See also: – **setObjectValue:** (NSControl)

selectItemWithObjectValue:

– (void)**selectItemWithObjectValue:**(id)*anObject*

Selects the first pop-up list item that corresponds to *anObject*. Objects are considered equal if they have the same **id** or if **isEqual:** returns YES. Posts NSComboBoxSelectionDidChangeNotification to the default notification center if the selection does in fact change. Note that this method doesn't alter the contents of the combo box's text field—see "Interacting with the Text Field" in the class description for more information.

See also: – **setObjectValue:** (NSControl)

setDataSource:

– (void)**setDataSource:**(id)*aSource*

Sets the receiver's data source to *aSource*. *aSource* should implement the appropriate methods of the NSComboBoxDataSource informal protocol. This method doesn't automatically set **usesDataSource** to NO, and in fact logs a warning if **usesDataSource** returns NO.

This method logs a warning if *aSource* doesn't respond to either **numberOfRowsInComboBox:** or **comboBox:objectValueForItemAtIndex:**.

See also: – **setUsesDataSource:**

setHasVerticalScroller:

– (void)**setHasVerticalScroller:**(BOOL)*flag*

Determines according to *flag* whether the receiver displays a vertical scroller. By default, *flag* is YES. If *flag* is NO and the combo box has more list items (either in its internal item list or from its data source) than are allowed by **numberOfVisibleItems**, only a subset will be displayed. NSComboBox's **scroll...** methods can be used to position this subset within the pop-up list.

Note that if *flag* is YES, a scroller will be displayed even if the combo box has fewer list items than are allowed by **numberOfVisibleItems**.

See also: – **numberOfItems**, **– scrollItemAtIndexToTop:**, **– scrollItemAtIndexToVisible:**

setIntercellSpacing:

– (void)**setIntercellSpacing:**(NSSize)*aSize*

Sets the width and height between pop-up list items to those in *aSize*. The default intercell spacing is (3.0, 2.0).

See also: – **setItemHeight:**, – **setNumberOfVisibleItems:**

setItemHeight:

– (void)**setItemHeight:**(float)*itemHeight*

Sets the height for items to *itemHeight*.

See also: – **setIntercellSpacing:**, – **setNumberOfVisibleItems:**

setNumberOfVisibleItems:

– (void)**setNumberOfVisibleItems:**(int)*visibleItems*

Sets the maximum number of items that will be visible at one time in the receiver's pop-up list to *visibleItems*.

See also: – **numberOfItems**, – **setItemHeight:**, – **setIntercellSpacing:**

setUsesDataSource:

– (void)**setUsesDataSource:**(BOOL)*flag*

Sets according to *flag* whether the receiver uses an external data source (specified by **setDataSource:**) to populate the receiver's pop-up list.

usesDataSource

– (BOOL)**usesDataSource**

Returns YES if the receiver uses an external data source to populate the receiver's pop-up list, NO if it uses an internal item list.

See also: – **dataSource**

Notifications

NSComboBoxSelectionDidChangeNotification

Posted after the NSComboBox's pop-up list selection changes. The notification contains:

Notification Object	The NSComboBox whose selection changed.
Userinfo	None

NSComboBoxCellSelectionIsChangingNotification

Posted whenever the NSComboBox's pop-up list selection is changing. The notification contains:

Notification Object	The NSComboBox whose selection is changing.
Userinfo	None

NSComboBoxCellWillPopUpNotification

Posted whenever the NSComboBox's pop-up list is going to be displayed. The notification contains:

Notification Object	The NSComboBox whose popup window will be displayed.
Userinfo	None

NSComboBoxCellWillDismissNotification

Posted whenever the NSComboBox's pop-up list is about to be dismissed. The notification contains:

Notification Object	The NSComboBox whose pop-up list will be dismissed.
Userinfo	None

NSComboBoxCell

Inherits From: NSTextFieldCell : NSActionCell : NSCell : NSObject

Conforms To: NSObject (from NSObject)
NSCoding (from NSCell)
NSCopying (from NSCell)

Declared In: AppKit/NSComboBoxCell.h

Class Description

NSComboBoxCell is a subclass of NSTextFieldCell used to implement the user interface of “combo boxes” (see the Class Description in the NSComboBox class specification for information on how combo boxes look and work). The NSComboBox subclass of NSTextField uses a single NSComboBoxCell, and essentially all of NSComboBox’s methods simply invoke the corresponding NSComboBoxCell method.

Method Types

Setting display attributes

– hasVerticalScroller
– intercellSpacing
– itemHeight
– numberOfVisibleItems
– setHasVerticalScroller:
– setIntercellSpacing:
– setItemHeight:
– setNumberOfVisibleItems:

Setting a data source

– dataSource
– setDataSource:
– setUsesDataSource:
– usesDataSource

Working with an internal list

– addItemsWithObjectValues:
– addItemWithObjectValue:
– insertItemWithObjectValue:atIndex:
– objectValues
– removeAllItems
– removeItemAtIndex:
– removeItemWithObjectValue:
– numberOfItems

Manipulating the displayed list

– indexOfItemWithObjectValue:
– itemObjectValueAtIndex:
– noteNumberOfItemsChanged
– reloadData
– scrollItemAtIndexToTop:
– scrollItemAtIndexToVisible:

Manipulating the selection

– deselectItemAtIndex:
– indexOfSelectedItem
– objectValueOfSelectedItem
– selectItemAtIndex:
– selectItemWithObjectValue:

Encoding a ComboBoxCell

– encodeWithCoder:
– initWithCoder:

Instance Methods

addItemsWithObjectValues:

– (void)**addItemsWithObjectValues:**(id)*objects*

Adds multiple objects to the end of the combo box cell's internal item list. This method logs a warning if **usesDataSource** returns YES.

addItemWithObjectValue:

– (void)**addItemWithObjectValue:**(id)*anObject*

Adds *anObject* to the end of the combo box cell's internal item list. This method logs a warning if **usesDataSource** returns YES.

dataSource

– (id)**dataSource**

Returns the object that provides the data displayed in the receiver's pop-up list. This method logs a warning if **usesDataSource** returns NO. See the class description and the NSComboBoxCellDataSource informal protocol specification for more information on combo box cell data source objects.

deselectItemAtIndex:

– (void)**deselectItemAtIndex:**(int)*index*

Deselects the pop-up list item at *index* if it's selected. If the selection does in fact change, this method posts an NSComboBoxSelectionDidChangeNotification to the default notification center.

See also: – **indexOfSelectedItem**, – **numberOfItems**, – **selectItemAtIndex:**

encodeWithCoder:

– (void)**encodeWithCoder:**(NSCoder *)*encoder*

Encodes the receiver using *encoder*. If the receiver uses a data source, the data source is conditionally encoded as well.

See also: – **initWithCoder:**

hasVerticalScroller

– (BOOL)**hasVerticalScroller**

Returns YES if the receiver will display a vertical scroller. Note that the scoller will be displayed even if the pop-up list contains fewer items than will fit in the area specified for display. Returns NO if the receiver won't display a vertical scroller.

See also: – **numberOfItems**, – **numberOfVisibleItems**

indexOfItemWithObjectValue:

– (int)**indexOfItemWithObjectValue:**(id)*anObject*

Searches the receiver's internal item list for *anObject* and returns the lowest index whose corresponding value is equal to *anObject*. Objects are considered equal if they have the same **id** or if **isEqual:** returns YES.

If none of the objects in the receiver's internal item list are equal to *anObject*, **indexOfItemObjectValue:** returns NSNotFound. This method logs a warning if **usesDataSource** returns YES.

See also: – **selectItemWithObjectValue:**

indexOfSelectedItem

– (int)**indexOfSelectedItem**

Returns the index of the last item selected from the receiver's pop-up list, or -1 if no item is selected. Note that nothing is initially selected in a newly-initialized combo box cell.

See also: – **objectValueOfSelectedItem**

initWithCoder:

– (id)**initWithCoder:**(NSCoder *)*decoder*

Initializes a newly-allocated instance from data in *decoder*. If the decoded instance uses a data source, **initWithCoder:** decodes the data source as well. Returns **self**.

See also: – **encodeWithCoder:**

insertItemWithObjectValue:atIndex:

– (void)**insertItemWithObjectValue:**(id)*anObject* **atIndex:**(int)*index*

Inserts *anObject* at *index* in the combo box cell's internal item list, shifting the previous item at *index*—along with all following items—down one slot to make room. This method logs a warning if **usesDataSource** returns YES.

See also: – **addItemWithObjectValue:**, – **numberOfItems**

intercellSpacing

– (NSSize)**intercellSpacing**

Returns the horizontal and vertical spacing between cells in the receiver's pop-up list. The default spacing is (3.0, 2.0).

See also: – **itemHeight**, – **numberOfVisibleItems**

itemHeight

– (float)**itemHeight**

Returns the height of each item in the receiver's pop-up list. The default item height is 16.0.

See also: – **intercellSpacing**, – **numberOfVisibleItems**

itemObjectValueAtIndex:

– (id)**itemObjectValueAtIndex:**(int)*index*

Returns the object located at *index* within the receiver's internal item list. If *index* is beyond the end of the list, an NSRangeException is raised. This method logs a warning if **usesDataSource** returns YES.

See also: – **objectValueOfSelectedItem**

noteNumberOfItemsChanged

– (void)**noteNumberOfItemsChanged**

Informs the receiver that the number of items in its data source has changed, allowing the receiver to update the scrollers in its displayed pop-up list without actually reloading data into the receiver. This method is particularly useful for a data source that continually receives data in the background over a period of time, in which case the NSComboBoxCell can remain responsive to the user while the data is received.

See the NSComboBoxCellDataSource informal protocol specification for information on the messages an NSComboBoxCell sends to its data source.

See also: – **reloadData**

numberOfItems

– (int)**numberOfItems**

Returns the total number of items in the pop-up list.

See also: – **numberOfItemsInComboBoxCell:** (NSComboBoxCellDataSource protocol),
– **numberOfVisibleItems**

numberOfVisibleItems

– (int)**numberOfVisibleItems**

Returns the maximum number of items visible at any one time in the pop-up list.

See also: – **numberOfItems**

objectValueOfSelectedItem

– (id)**objectValueOfSelectedItem**

Returns the object from the receiver's internal item list corresponding to the last item selected from the pop-up list, or **nil** if no item is selected. Note that nothing is initially selected in a newly-initialized combo box cell. This method logs a warning if **usesDataSource** returns YES.

See also: – **comboBoxCell:objectValueForItemAtIndex:** (NSComboBoxCellDataSource protocol),
– **indexOfSelectedItem**

objectValues

– (NSArray *)**objectValues**

Returns as an array the receiver's internal item list. This method logs a warning if **usesDataSource** returns YES.

reloadData

– (void)**reloadData**

Marks the receiver as needing redisplay, so that it will reload the data for visible pop-up items and draw the new values.

See also: – **noteNumberOfItemsChanged**

removeAllItems

– (void)**removeAllItems**

Removes all items from the receiver's internal item list. This method logs a warning if **usesDataSource** returns YES.

See also: – **objectValues**

removeItemAtIndex:

– (void)**removeItemAtIndex:**(int)*index*

Removes the object at *index* from the receiver's internal item list and moves all items beyond *index* up one slot to fill the gap. The removed object receives a release message. This method raises an NSRangeException if *index* is beyond the end of the list, and logs a warning if **usesDataSource** returns YES.

removeItemWithObjectValue:

– (void)**removeItemWithObjectValue:**(id)*anObject*

Removes all occurrences of *anObject* from the receiver's internal item list. Objects are considered equal if they have the same **id** or if **isEqual:** returns YES. This method logs a warning if **usesDataSource** returns YES.

See also: – **indexOfItemWithObjectValue:**

scrollItemAtIndexToTop:

– (void)**scrollItemAtIndexToTop:**(int)*index*

Scrolls the receiver's pop-up list vertically so that the item specified by *index* is as close to the top as possible. The pop-up list need not be displayed at the time this method is invoked.

scrollItemAtIndexToVisible:

– (void)**scrollItemAtIndexToVisible:**(int)*index*

Scrolls the receiver's pop-up list vertically so that the item specified by *index* is visible. The pop-up list need not be displayed at the time this method is invoked.

selectItemAtIndex:

– (void)**selectItemAtIndex:**(int)*index*

Selects the pop-up list row at *index*. Posts NSComboBoxSelectionDidChangeNotification to the default notification center if the selection does in fact change. Note that this method does not alter the contents of the combo box cell's text field—see "Interacting with the Text Field" in the class description for more information.

See also: – **setObjectValue:** (NSControl)

selectItemWithObjectValue:

– (void)**selectItemWithObjectValue:**(id)*anObject*

Selects the first pop-up list item that corresponds to *anObject*. Objects are considered equal if they have the same **id** or if **isEqual:** returns YES. Posts NSComboBoxSelectionDidChangeNotification to the default notification center if the selection does in fact change. Note that this method doesn't alter the contents of the combo box cell's text field—see "Interacting with the Text Field" in the class description for more information.

See also: – **setObjectValue:** (NSControl)

setDataSource:

– (void)**setDataSource:**(id)*aSource*

Sets the receiver's data source to *aSource*. *aSource* should implement the appropriate methods of the NSComboBoxCellDataSource informal protocol. This method doesn't automatically set **usesDataSource** to NO, and in fact logs a warning if **usesDataSource** returns NO.

This method logs a warning if *aSource* doesn't respond to either **numberOfRowsInComboBoxCell:** or **comboBoxCell:objectValueForItemAtIndex:**.

See also: – **setUsesDataSource:**

setHasVerticalScroller:

– (void)**setHasVerticalScroller:**(BOOL)*flag*

Determines according to *flag* whether the receiver displays a vertical scroller. By default, *flag* is YES. If *flag* is NO and the combo box cell has more list items (either in its internal item list or from its data source) than are allowed by **numberOfVisibleItems**, only a subset will be displayed. NSComboBoxCell's **scroll...** methods can be used to position this subset within the pop-up list.

Note that if *flag* is YES, a scroller will be displayed even if the combo box cell has fewer list items than are allowed by **numberOfVisibleItems**.

See also: – **numberOfItems**, **– scrollItemAtIndexToTop:**, **– scrollItemAtIndexToVisible:**

setIntercellSpacing:

– (void)**setIntercellSpacing:**(NSSize)*aSize*

Sets the width and height between pop-up list items to those in *aSize*. The default intercell spacing is (3.0, 2.0).

See also: – **setItemHeight:**, – **setNumberOfVisibleItems:**

setItemHeight:

– (void)**setItemHeight:**(float)*itemHeight*

Sets the height for items to *itemHeight*.

See also: – **setIntercellSpacing:**, – **setNumberOfVisibleItems:**

setNumberOfVisibleItems:

– (void)**setNumberOfVisibleItems:**(int)*visibleItems*

Sets the maximum number of items that will be visible at one time in the receiver's pop-up list to *visibleItems*.

See also: – **numberOfItems**, – **setItemHeight:**, – **setIntercellSpacing:**

setUsesDataSource:

– (void)**setUsesDataSource:**(BOOL)*flag*

Sets according to *flag* whether the receiver uses an external data source (specified by **setDataSource:**) to populate the receiver's pop-up list.

usesDataSource

– (BOOL)**usesDataSource**

Returns YES if the receiver uses an external data source to populate the receiver's pop-up list, NO if it uses an internal item list.

See also: – **dataSource**

NSControl

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

Class Description

NSControl is an abstract superclass that provides three fundamental features for implementing user-interface devices. First, as a subclass of NSView, NSControl draws, or coordinates the drawing of, the on-screen representation of the device. Second, it receives and responds to user-generated events within its bounds by overriding NSResponder's **mouseDown:** method and providing a position in the responder chain. Third, it implements the **sendAction:to:** method to send an action message to the NSControl's target object. Subclasses of NSControl defined in the Application Kit are NSBrowser, NSButton (and its subclass NSPopUpButton), NSColorWell, NSImageView, NSMatrix (and its subclass NSForm), NSScroller, NSSlider, NSTableView, and NSTextField. Instances of concrete NSControl subclasses are often referred to as, simply, *controls*.

Controls and Cells

Controls are usually associated with one or more cells—instances of a subclass of the abstract class NSCell. A control's cell (or cells) usually fit just inside the bounds of the control. Cells are objects that can draw themselves and respond to events, but they can do so only indirectly, upon instruction from their control, which acts as a kind of coordinating backdrop.

Controls manage the behavior of their cells. By inheritance from NSView, controls derive the ability for responding to user actions and rendering their on-screen representation. When users click on a control, it responds in part by sending **trackMouse:inRect:ofView:untilMouseUp:** to the cell that was clicked; upon receiving this message, the cell tracks the mouse and may have the control send the cell's action message to its target (either upon mouse-up or continuously, depending on the cell's attributes). When controls receive a display request, they, in turn, send their cell (or cells) a **drawWithFrame:inView:** message to have the cells draw themselves.

This relationship of control and cell makes two things possible: A control can manage cells of different types and with different targets and actions (see below); and a single control can manage muliple cells. Most Application Kit controls, like NSButtons and NSTextFields, manage only a single cell. But some controls, notably NSMatrix and NSForm, manage multiple cells (usually of the same size and attributes, and arranged in a regular pattern). Because cells are lighter-weight than controls, in terms of inherited data and behavior, it is more efficient to use a multi-cell control rather than multiple controls.

Many methods of NSControl—particularly methods that set or obtain values and attributes—have corresponding methods in NSCell. Sending a message to the control causes it to be forwarded to the control's cell *or* (if a multi-cell control) its selected cell. However, many NSControl methods are effective only in controls with single cells (these are noted in the method descriptions).

An NSControl subclass doesn't have to use an NSCell subclass to implement itself; NSScroller and NSColorWell are examples of NSControls that don't. However, such subclasses have to take care of details that NSCell would otherwise handle. Specifically, they have to override methods designed to work with a cell. What's more, the lack of a cell means you can't make use of NSMatrix capability for managing multi-cell arrays such as radio buttons.

Target and Action

Target objects and action methods (or messages) are part of the mechanism by which controls respond to user actions and enable users to communicate their intentions to an application. A target is an object that a control uses as the receiver of action messages. The target's class defines an action method to enable its instances to respond to these messages, which are sent as users click or otherwise manipulate the control. NSControl's **sendAction:to:** asks the NSApplication object, NSApp, to send an action message to the control's target object.

NSControl provides methods for setting and obtaining the target object and the action method. However, these methods require that an NSControl's cell (or cells) be cells that inherit from NSActionCell or custom cells that hold action and target as instance variables and can respond to the NSControl methods.

See the NSActionCell class specification for more on the implementation of target and action behavior, particularly how action messages with **nil** targets travel up the responder chain.

Field Validation and Entry Error-Handling

NSControl provides the delegation method **control:isValidObject:** for validating the contents of cells embedded in controls (instances of NSTextField and NSMatrix in particular). In validating you check for values that are permissible as objects, but that are undesirable in a given context, such as a date field in which dates should never be in the future, or zip codes that are valid for a certain state.

The method **control:isValidObject:** is invoked when the insertion point leaves a cell (that is, the associated control relinquishes first-responder status) but before the string value of the cell's object is displayed. Return YES to allow display of the string and NO to reject display and return the cursor to the cell. The following example evaluates an object (an NSDate) and rejects it if the date is in the future:

```
- (BOOL)control:(NSControl *)control isValidObject:(id)obj
{
    if (control == contactsForm) {
        if (![obj isKindOfClass:[NSDate class]]) return NO;
        if ([[obj laterDate:[NSDate date]] isEqual:obj]) {
            NSRunAlertPanel(@"Date not valid",
                @"Reason: date in future", NULL, NULL, NULL);
            return NO;
        }
    }
    return YES;
}
```

NSControl provides several delegate methods for its subclasses that allow text editing, such as NSTextField and NSMatrix. Some are invoked when formatters for a control's cells cannot format a string **(control: didFailToFormatString:errorDescription:**) or reject a partial string entry (**control: didFailToValidatePartialString:errorDescription:**). NSControl also provides **control:textView: doCommandBySelector:**, which allows delegates the opportunity to detect and respond to key bindings, such as **complete:** (name completion). Note that although NSControl defines delegate methods, it does not itself have a delegate. Any subclass that uses the delegate methods must contain a delegate and the methods to get and set it.

Changing the NSCell Class

Since NSControl uses objects derived from the NSCell class to implement most of its actual functionality, you can usually implement a unique user interface device by creating a subclass of NSCell rather than NSControl. As an example, let's say you want all your application's NSSliders to have a type of cell other than the generic NSSliderCell. First, you create a subclass of NSCell, NSActionCell, or NSSliderCell. (Let's call it MyCellSubclass.) Then, you can simply invoke NSSlider's **setCellClass:** class method:

```
[NSSlider setCellClass:[MyCellSubclass class]];
```

All NSSliders created thereafter will use MyCellSubclass, until you call **setCellClass:** again.

If you want to create generic NSSliders (ones that use NSSliderCell) in the same application as the customized NSSliders that use MyCellSubclass, there are two possible approaches. One is to invoke **setCellClass:** as above whenever you're about to create a custom NSSlider, resetting the cell class to NSSliderCell afterwards. The other approach is to create a custom subclass of NSSlider that automatically uses MyCellSubclass, as explained below.

Creating New NSControls

If you create a custom NSControl subclass that uses a custom subclass of NSCell, you should override NSControl**'s cellClass** method:

```
+ (Class) cellClass
{
    return [MyCellSubclass class];
}
```

NSControl's **initWithFrame:** method will use the return value of **cellClass** to allocate and initialize an NSCell of the correct type.

Override the designated initializer (**initWithFrame:**) if you create a subclass of NSControl that performs its own initialization.

Method Types

Initializing an NSControl

– initWithFrame:

Setting the control's cell

+ cellClass
+ setCellClass:
– cell
– setCell:

Enabling and disabling the control

– isEnabled
– setEnabled:

Identifying the selected cell

– selectedCell
– selectedTag

Setting the control's value

– doubleValue
– setDoubleValue:
– floatValue
– setFloatValue:
– intValue
– setIntValue:
– objectValue
– setObjectValue:
– stringValue
– setStringValue:
– setNeedsDisplay

Interacting with other controls

– takeDoubleValueFrom:
– takeFloatValueFrom:
– takeIntValueFrom:
– takeObjectValueFrom:
– takeStringValueFrom:

Formating text

– alignment
– setAlignment:
– font
– setFont:
– setFloatingPointFormat:left:right:

Managing the field editor

– abortEditing
– currentEditor
– validateEditing

Resizing the control

– calcSize
– sizeToFit

Displaying a cell

– selectCell:
– drawCell:
– drawCellInside:
– updateCell:
– updateCellInside:

Implementing the target/action mechanism

– action
– setAction:
– target
– setTarget:
– isContinuous
– setContinuous:
– sendAction:to:
– sendActionOn:

Getting and setting attributed-string values

– attributedStringValue
– setAttributedStringValue:

Getting and setting tags

– tag
– setTag:

Activating from the keyboard

– performClick:
– refusesFirstResponder
– setRefusesFirstResponder:

Tracking the mouse

– mouseDown:
– ignoresMultiClick
– setIgnoresMultiClick:

Class Methods

cellClass

+ (Class)**cellClass**

Returns the class of cells used by the receiving class (which must be NSControl or one of its subclasses). Returns **nil** if no cell class has been specified for the receiving class or any of its superclasses (up to NSControl).

See also: – **cell**, – **setCell:**

setCellClass:

+ (void)**setCellClass:**(Class)*class*

Sets the class of cells used by instances of the receiver, which must be the NSControl class or one of its subclasses.

See also: – **cell**, – **setCell:**

Instance Methods

abortEditing

– (BOOL)**abortEditing**

Terminates and discards any editing of text displayed by the receiving control and removes the field editor's delegate. Returns YES if there was a field editor associated with the control, NO otherwise.

See also: – **currentEditor**, – **validateEditing**

action

– (SEL)**action**

Returns the action-message selector of the receiver's cell (the default NSControl behavior), or the default action-message selector for a control with multiple cells (such as an NSMatrix or an NSForm). For controls with multiple cells, it's better to get the action-message selector for a particular cell, for instance:

```
someAction = [[theControl selectedCell] action];
```

See also: – **setAction:**, – **setTarget:**, – **target**

alignment

– (NSTextAlignment)**alignment**

Returns the alignment mode of the text in the receiver's cell. The return value can be one of these constants: NSLeftTextAlignment, NSRightTextAlignment, NSCenterTextAlignment, NSJustifiedTextAlignment, or NSNaturalTextAlignment (the default alignment).

See also: – **setAlignment:**

attributedStringValue

– (NSAttributedString *)**attributedStringValue**

Returns the object value of the receiver’s cell (or selected cell) as an attributed string after validating any editing currently being done. If no cell is associated with the receiver, returns an empty attributed string.

See also: – **setAttributedStringValue:**

calcSize

– (void)**calcSize**

Recomputes any internal sizing information for the NSControl, if necessary, by invoking its NSCell’s **calcDrawInfo:** method. Most NSControls maintain a flag that informs them if any of their cells have been modified in such a way that the location or size of the cell should be recomputed. If this happens, **calcSize** is automatically invoked whenever the NSControl is displayed; you never need to invoke it yourself.

See also: – **sizeToFit**

cell

– (id)**cell**

Returns the receiver's cell. In NSControls with multiple cells (such as NSMatrix or NSForm), use **selectedCell** or a similar method for finding a particular cell.

See also: + **cellClass**, – **setCell:**, + **setCellClass:**

currentEditor

– (NSText *)**currentEditor**

If the receiving NSControl is being edited—that is, it has an NSText object acting as its field editor, and is the first responder of its NSWindow—this method returns the NSText editor; otherwise, it returns **nil**.

See also: – **abortEditing**, – **validateEditing**

doubleValue

– (double)**doubleValue**

Returns the value of the receiver's cell as a double-precision floating point number. If the NSControl contains many cells (for example, NSMatrix), then the value of the currently **selectedCell** is returned. If the NSControl is in the process of editing the affected Cell, then **validateEditing** is invoked before the value is extracted and returned.

See also: – **floatValue**, – **intValue**, – **objectValue**, – **setDoubleValue:**, – **stringValue**

drawCell:

– (void)**drawCell:**(NSCell *)*aCell*

If *aCell* is the cell used to implement this NSControl, then the NSControl is displayed. This method is provided primarily to support a consistent set of methods between NSControls with single and multiple cells, since a NSControl with multiple cells needs to be able to draw a single cell at a time.

See also: – **selectCell:**, – **updateCell:**, – **updateCellInside:**

drawCellInside:

– (void)**drawCellInside:**(NSCell *)*aCell*

Draws the inside of the receiver's cell (the area within a bezel or border). If the NSControl is transparent, the method causes the superview to draw itself. This method invokes NSCell's **drawInteriorWithFrame:**

inView: method. This method has no effect on NSControls (such as NSMatrix and NSForm) that have multiple cells.

See also: – **selectCell:**, – **updateCell:**, – **updateCellInside:**

floatValue

– (float)**floatValue**

Returns the value of the receiver's cell (or selected cell, if a multiple-cell NSControl) as a single-precision floating point number. See **doubleValue** for more details.

See also: – **doubleValue**, – **intValue**, – **objectValue**, – **setFloatValue:**, – **stringValue**

font

– (NSFont *)**font**

Returns the NSFont used to draw text in the receiver’s cell.

See also: – **setFont:**

ignoresMultiClick

– (BOOL)**ignoresMultiClick**

Returns whether the receiving NSControl ignores multiple clicks made in rapid succession. See **setIgnoresMultiClick:** for details.

initWithFrame:

– (id)**initWithFrame:**(NSRect)*frameRect*

Initializes and returns a new NSControl object in *frameRect*, and creates a cell for it if the cell’s class has been specified for controls of this type with **setCellClass:**. Because NSControl is an abstract class, invocations of this method should appear only in the designated initializers of sublcasses; that is, there should always be a more specific designated initializer for the subclass, as this **initWithFrame:** is the designated initializer for NSControl.

intValue

– (int)**intValue**

Returns the value of the receiver's cell (or selected cell, if a multiple-cell NSControl) as an integer. See **doubleValue** for more details.

See also: – **floatValue**, – **doubleValue**, – **objectValue**, – **setIntValue:**, – **stringValue**

isContinuous

– (BOOL)**isContinuous**

Returns whether the control’s NSCell continuously sends its action message to its target during mouse tracking.

See also: – **setContinuous:**

isEnabled

– (BOOL)**isEnabled**

Returns whether the receiver reacts to mouse events.

See also: – **setEnabled:**

mouseDown:

– (void)**mouseDown:**(NSEvent *)*theEvent*

Invoked when the mouse button is pressed while the cursor is within the bounds of the NSControl. This method highlights the NSControl’s NSCell and sends it a **trackMouse:inRect:ofView:untilMouseUp:** message. Whenever the NSCell finishes tracking the mouse (for example, because the cursor has left the cell’s bounds), the cell is unhighlighted. If the mouse button is still down and the cursor reenters the bounds, the cell is again highlighted and a new **trackMouse:inRect:ofView:untilMouseUp:** message is sent. This behavior repeats until the mouse button goes up. If it goes up with the cursor in the control, the state of the control is changed, and the action message is sent to the target. If the mouse button goes up when the cursor is outside the control, no action message is sent.

See also: – **ignoresMultiClick**, – **trackMouse:inRect:ofView:untilMouseUp:**(NSCell)

objectValue

– (id)**objectValue**

Returns the value of the receiver's cell (or selected cell, if a multiple-cell NSControl) as an Objective-C object. See **doubleValue** for more details.

See also: – **floatValue**, – **doubleValue**, – **intValue**, – **setObjectValue:**, – **stringValue**

performClick:

– (void)**performClick:***sender*

Programmatically simulates a mouse click on the receiver’s cell, including the invocation of the action method in the target object. Raises an exception if the action message cannot be successfully sent.

refusesFirstResponder

– (BOOL)**refusesFirstResponder**

Returns whether the receiver refuses first responder status.

See also: – **setRefusesFirstResponder:**

selectCell:

– (void)**selectCell:**(NSCell *)*aCell*

If *aCell* is a cell of the receiving NSControl and is unselected, this method selects *aCell* (by setting its state to YES) and redraws the NSControl.

See also: – **selectedCell**

selectedCell

– (id)**selectedCell**

Returns the receiver’s selected cell. The default implementation for NSControl simply returns the associated cell (or **nil** if no cell has been set). Subclasses of NSControl that manage multiple cells (such as NSMatrix and NSForm) override this method to return the cell selected by users.

See also: – **cell**, – **setCell:**

selectedTag

– (int)**selectedTag**

Returns the tag integer of the receiver's selected cell (see **selectedCell**) or -1 if there is no selected cell. When you set the tag of an control with a single cell in Interface Builder, it sets the tags of both the control and the cell with the same value as a convenience.

See also: – **setTag:**, – **tag**

sendAction:to:

– (BOOL)**sendAction:**(SEL)*theAction* **to:**(id)*theTarget*

Sends **sendAction:to:from:** to NXApp, which in turn sends a message to *theTarget* to perform *theAction*, adding the receiver as the argument to the **from:** keyword. **sendAction:to:** is invoked primarily by NSCell's **trackMouse:inRect:ofView:untilMouseUp:**.

If *theAction* is **nil**, no message is sent. If *theTarget* is **nil**, NXApp looks for an object that can respond to the message by following the responder chain (see the class description for NSActionCell). This method returns **nil** if no object that responds to *theAction* could be found.

See also: – **action**, – **target**

sendActionOn:

– (int)**sendActionOn:**(int)*mask*

Sets the conditions on which the receiver sends action messages to its target (continuously, mouse up, and others) and returns a bit mask with which to detect the previous settings. NSControl's default implementation simply invokes the **sendActionOn:** method of its associated cell

See also: – **sendAction:to:**, – **sendActionOn:**(NSCell)

setAction:

– (void)**setAction:**(SEL)*aSelector*

Sets the NSControl's action method to *aSelector*. If *aSelector* is **nil**, then no action messages will be sent from the NSControl.

See also: – **action**, – **setTarget:**, – **target**

setAlignment:

– (void)**setAlignment:**(NSTextAlignment)*mode*

Sets the alignment of text in the receiver's cell and, if the cell is being edited, aborts editing and updates the cell. *mode* is one of five constants: NSLeftTextAlignment, NSRightTextAlignment, NSCenterTextAlignment, NSJustifiedTextAlignment, NSNaturalTextAlignment (the default alignment for the text).

See also: – **alignment**

setAttributedStringValue:

– (void)**setAttributedStringValue:**(NSAttributedString *)*object*

Sets the value of the receiver's cell (or selected cell) as an attributed string. If the cell is being edited, it aborts all editing before setting the value; if the cell doesn't inherit from NSActionCell, it marks it for automatic redisplay (NSActionCell performs its own updating of cells).

See also: – **attributedStringValue**

setCell:

– (void)**setCell:**(NSCell *)*aCell*

Sets the receiver's cell to *aCell*. Use this method with great care as it can irrevocably damage the affected control; specifically, you should only use this method in initializers for subclasses of NSControl.

See also: – **cell**, – **selectedCell**

setContinuous:

– (void)**setContinuous:**(BOOL)*flag*

Sets whether the receiver's cell continuously sends its action message to its target as it tracks the mouse.

See also: – **isContinuous**

setDoubleValue:

– (void)**setDoubleValue:**(double)*aDouble*

Sets the value of the receiver's cell (or selected cell) as aDouble (a double-precision floating point number). If the cell is being edited, it aborts all editing before setting the value; if the cell doesn't inherit from

NSActionCell, it marks the cell's interior for automatic redisplay (NSActionCell performs its own updating of cells).

See also: – **doubleValue**,– **setFloatValue:**, – **setIntValue:**, – **setObjectValue:**, – **setStringValue:**

setEnabled:

– (void)**setEnabled:**(BOOL)*flag*

Sets whether the receiving NSControl's cell—or if there is no associated cell, the NSControl itself—is active (that is, whether it tracks the mouse and sends its action to its target). If flag is NO, any editing is aborted. Redraws the entire Control if autodisplay is enabled. Subclasses may want to override this method to redraw only a portion of the control when the enabled state changes, as do NSButton and NSSlider.

See also: – **isEnabled**

setFloatValue:

– (void)**setFloatValue:**(float)*aFloat*

Sets the value of the receiver's cell (or selected cell) as *aFloat* (a single-precision floating point number). If the cell is being edited, it aborts all editing before setting the value; if the cell doesn't inherit from NSActionCell, it marks the cell's interior for automatic redisplay (NSActionCell performs its own updating of cells).

See also: – **floatValue**, – **setDoubleValue:**, – **setIntValue:**, – **setObjectValue:**, – **setStringValue:**

setFloatingPointFormat:left:right:

– (void)**setFloatingPointFormat:**(BOOL)*autoRange*
left:(unsigned)*leftDigits*
right:(unsigned)*rightDigits*

Sets the autoranging and floating point number format of the receiver's cell, so that at most *leftDigits* are displayed to the left of the decimal point, and *rightDigits* to the right. See the description of this method in the NSCell class specification for details. If the cell is being edited, what's typed is discarded and the cell's interior is redrawn.

See also: – **setFloatingPointFormat:left:right:**(NSCell)

setFont:

– (void)**setFont:**(NSFont *)*fontObject*

Sets the font used to draw text in the receiver's cell to *fontObject*. If the cell is being edited, the text in the cell is redrawn in the new font and the cell's editor (the NSText object used globally for editing) is updated with the new NSFont.

See also: – **setFont:**

setIgnoresMultiClick:

– (void)**setIgnoresMultiClick:**(BOOL)*flag*

Sets whether the receiving NSControl ignores multiple clicks made in rapid succession. By default, controls treat double-clicks as two distinct clicks, triple-clicks as three distinct clicks, and so on. However, when an NSControl returning YES to this method receives multiple clicks (within a predetermined interval), each **mouseDown** event after the first is passed on to **super**.

See also: – **ignoresMultiClick**

setIntValue:

– (void)**setIntValue:**(int)*anInt*

Sets the value of the receiver's cell (or selected cell) as an integer (*anInt*). If the cell is being edited, it aborts all editing before setting the value; if the cell doesn't inherit from NSActionCell, it marks the cell's interior for automatic redisplay (NSActionCell performs its own updating of cells).

See also: – **intValue**,– **setDoubleValue:**, – **setFloatValue:**, – **setObjectValue:**, – **setStringValue:**

setNeedsDisplay

– (void)**setNeedsDisplay**

Marks the receiving NSControl as needing redisplay (assuming automatic display is enabled) after recalculation of its dimensions.

See also: – **setNeedsDisplay:** (NSView)

setObjectValue:

– (void)**setObjectValue:**(id)*object*

Sets the value of the receiver's cell (or selected cell) as an Objective-C object. If the cell is being edited, it aborts all editing before setting the value; if the cell doesn't inherit from NSActionCell, it marks the cell's interior for automatic redisplay (NSActionCell performs its own updating of cells).

See also: – **objectValue**, – **setDoubleValue:**, – **setFloatValue:**, – **setIntValue:**, – **setStringValue:**

setRefusesFirstResponder:

– (void)**setRefusesFirstResponder:**(BOOL)*flag*

Sets whether the receiver refuses first responder status. By default, the user can advance the focus of keyboard events between controls by pressing the Tab key; when this focus—or first responder status— is indicated for a control (by the insertion point or, for non-text controls, a faint rectangle), the user can activate the control by pressing the space bar.

See also: – **refusesFirstResponder**, – **objectValue**, – **setDoubleValue:**, – **setFloatValue:**

setStringValue:

– (void)**setStringValue:**(NSString *)*aString*

Sets the value of the receiver's cell (or selected cell) as an NSString object (*aString*). If the cell is being edited, it aborts all editing before setting the value; if the cell doesn't inherit from NSActionCell, it marks the cell's interior for automatic redisplay (NSActionCell performs its own updating of cells).

See also: – **setDoubleValue:**, – **setFloatValue:**, – **setIntValue:**, – **setObjectValue:**, – **stringValue**

setTag:

– (void)**setTag:**(int)*anInt*

Sets the tag of the receiving NSControl to *anInt*. It doesn't affect the tag of the receiver's cell.

See also: – **tag**

setTarget:

– (void)**setTarget:**(id)*anObject*

Sets the target object for the action message of the receiver's cell; NSCell's **setTarget:** is used instead of any subclass override of this method. If *anObject* is **nil** and the control sends an action message, the

application looks for an object that can respond to the message by following the responder chain (see description of the NSActionCell class for details).

See also: – **action**, – **setAction:**, – **target**, – **setTarget:**(NSCell)

sizeToFit

– (void)**sizeToFit**

Changes the width and the height of the receiver's frame so that they are the minimum needed to contain its cell. If you want a multiple-cell custom subclass of NSControl to size itself to fit its cells, you must override this method.

See also: – **calcSize**

stringValue

– (NSString *)**stringValue**

Returns the value of the receiver's cell (or selected cell, if a multiple-cell NSControl) as an NSString object. See **doubleValue** for details.

See also: – **floatValue**, – **doubleValue**, – **intValue**, – **objectValue**, – **setStringValue:**

tag

– (int)**tag**

Returns the tag identifying the receiving control (not the tag of the receiver’s cell).

See also: – **setTag:**

takeDoubleValueFrom:

– (void)**takeDoubleValueFrom:**(id)*sender*

Sets the double-precision floating-point value of the receiving control's cell (or selected cell) to the value obtained by sending a **doubleValue** message to *sender*. You can use this method to link action messages between controls. It permits one control or cell (*sender*) to affect the value of another control (the receiver) by invoking this method in an action message to the receiver. For example, a text field can be made the target of a slider. Whenever the slider is moved, it will send a **takeDoubleValueFrom:** message to the text field. The text field will then get the slider's floating-point value, turn it into a text string, and display it, thus tracking the value of the slider.

takeFloatValueFrom:

– (void)**takeFloatValueFrom:**(id)*sender*

Sets the receiving NSControl's selected cell to the value obtained by sending a **floatValue** message to another control or cell (*sender*). See **takeDoubleValueFrom:** for more information.

takeIntValueFrom:

– (void)**takeIntValueFrom:**(id)*sender*

Sets the receiving NSControl's selected cell to the value obtained by sending a **intValue** message to another control or cell (*sender*). See **takeDoubleValueFrom:** for more information.

takeObjectValueFrom:

– (void)**takeObjectValueFrom:**(id)*sender*

Sets the receiving NSControl's selected cell to the value obtained by sending a **objectValue** message to another control or cell (*sender*). See **takeDoubleValueFrom:** for more information.

takeStringValueFrom:

– (void)**takeStringValueFrom:**(id)*sender*

Sets the receiving NSControl's selected cell to the value obtained by sending a **stringValue** message to another control or cell (*sender*). See **takeDoubleValueFrom:** for more information.

target

– (id)**target**

Returns the target object of the receiver’s cell.

See also: – **action**, – **setAction:**, – **setTarget:**

updateCell:

– (void)**updateCell:**(NSCell *)*aCell*

Redisplays *aCell* or marks it for redisplay.

updateCellInside:

– (void)**updateCellInside:**(NSCell *)*aCell*

Redisplays the inside of *aCell* or marks it for redisplay.

validateEditing

– (void)**validateEditing**

Validates the user's changes to text in a cell of the receiving control. Validation sets the object value of the cell to the current contents of the cell's editor (the NSText object used for editing), storing its a simple NSString or an attributed string object based on the attributes of the editor.

See also: – **abortEditing**, – **currentEditor**

Methods Implemented By the Delegate

NSControl provides several delegate methods for its subclasses that allow text editing, such as NSTextField and NSMatrix. Note that although NSControl defines delegate methods, it does not itself have a delegate. Any subclass that uses these methods must have a delegate and the methods to get and set it.

control:didFailToFormatString:errorDescription:

– (BOOL)**control:**(NSControl *)*control*
didFailToFormatString:(NSString *)*string*
errorDescription:(NSString *)*error*

Invoked when the formatter for *control*'s cell (or selected cell) cannot convert an NSString (*string)* to an underlying object. *error* is a localized user-presentable NSString that explains why the conversion failed. Evaluate the error or query the user and return YES if *string* should be accepted as-is, or NO if *string* should be rejected.

See also: – **getObjectValue:forString:errorDescription:**(NSFormatter)

control:didFailToValidatePartialString:errorDescription:

– (void)**control:**(NSControl *)*control*
didFailToValidatePartialString:(NSString *)*string*
errorDescription:(NSString *)*error*

Invoked when the formatter for *control*'s cell (or selected cell) rejects a partial string a user is typing into the cell. This NSString (*string*) includes the character that caused the rejection. *error* is a localized user-presentable NSString that explains why the validation failed. You can implement this method to

display a warning message or perform a similar action when the user enters enters improperly formatted text.

See also: – **isPartialStringValid:newEditingString:errorDescription:**(NSFormatter)

control:isValidObject:

– (BOOL)**control:**(NSControl *)*control* **isValidObject:**(id)*object*

Invoked when the insertion point leaves a cell but before the string value of the cell's object is displayed. Return YES to allow display of the string and NO to reject display and return the cursor to the cell. This method gives the delegate the opportunity to validate the contents of *control*'s cell (or selected cell). In validating, the delegate checks *object* to determine if it falls within a permissible range, has required attributes, accords with a given context, and so on. An example of an object subject to such and evaluation is an NSDate object which should not represent a future date, or a monetary amount (represented by an NSNumber) that exceeds a predetermined limit.

control:textShouldBeginEditing:

– (BOOL)**control:**(NSControl *)*control* **textShouldBeginEditing:**(NSText *)*fieldEditor*

Sent directly by *control* to the delegate when the user tries to enter a character in a cell of a control that allows editing of text (such as a text field or form field). Return YES if the NSControl's *fieldEditor* should be allowed to start editing the text, NO otherwise.

control:textShouldEndEditing:

– (BOOL)**control:**(NSControl *)*control* **textShouldEndEditing:**(NSText *)*fieldEditor*

Sent directly by *control* to the delegate when the insertion point tries to leave a cell of the control that has been edited. It's sent only by controls that allow editing of text (such as a text field or a form field). Return YES if the control's *fieldEditor* should be allowed to end its edit session, NO otherwise.

control:textView:doCommandBySelector:

– (BOOL)**control:**(NSControl *)*control*
textView:(NSTextView *)*textView*
doCommandBySelector:(SEL)*command*

Invoked when users press keys with predefined bindings in *control*'s cell or selected cell, as communicated to the control by the cell's field editor (*textView*). The delegate returns YES if it handles the key binding, and NO otherwise. These bindings are usually implemented as methods (*command*) defined in NSResponder; examples of such key bindings are arrow keys (for directional movement) and the Escape

key (for name completion). By implementing this method, the delegate can override the default implementation of *command* and supply its own behavior.

For example, the default method for completing partially typed path names or symbols (usually when users press the Escape key) is **complete:**. The default implementation of **complete:** (in NSResponder) does nothing. The delegate could evaluate *command* and, if it's **complete:**, get the current string from *textView* and then expand it, or display a list of potential completions, or do whatever else is appropriate.

controlTextDidBeginEditing:

– (void)**controlTextDidBeginEditing:**(NSNotification *)*aNotification*

Sent by the default notification center to the delegate and all observers of the notification when a control with editable cells (such as a text field, form field, or an NSMatrix) begins editing text. The name of the notification (*aNotification*) is always NSControlTextDidBeginEditingNotification. Use the key @"NSFieldEditor" to obtain the field editor from *aNotification*'s **userInfo** dictionary. If the delegate implements this method, it's automatically registered to receive this notification.

controlTextDidEndEditing:

– (void)**controlTextDidEndEditing:**(NSNotification *)*aNotification*

Sent by the default notification center to the delegate and all observers of the notification when a control with editable cells (such as a text field, form field, or an NSMatrix) ends editing text. The name of the notification (*aNotification*) is always NSControlTextDidEndEditingNotification. Use the key @"NSFieldEditor" to obtain the field editor from *aNotification*'s **userInfo** dictionary. If the delegate implements this method, it's automatically registered to receive this notification.

controlTextDidChange:

– (void)**controlTextDidChange:**(NSNotification *)*aNotification*

Sent by the default notification center to the delegate when the text in the receiving control (usually a text field, form, or NSMatrix with editable cells) changes. The name of the notification *aNotification* is always NSControlTextDidChangeNotification. Use the key @"NSFieldEditor" to obtain the field editor from *aNotification*'s **userInfo** dictionary. If the delegate implements this method, it's automatically registered to receive this notification.

Notifications

NSControl posts the following notifications to interested observers and its delegate.

Note that although NSControl defines delegate methods, it does not itself have a delegate. Any subclass that uses these methods must have a delegate and the methods to get and set it.

NSControlTextDidBeginEditingNotification

This notification object contains a notification object and a userInfo dictionary. The notification object is the NSControl posting the notification. (The field editor of the edited cell originally sends a NSTextDidBeginEditingNotification to the control, which passes it on in this form to its delegate.) The userInfo dictionary contains these keys and values:

Key	Value
@"NSFieldEditor"	The edited cell's field editor

See description of **controlTextDidBeginEditing:**, above, for details.

NSControlTextDidChangeNotification

This notification object contains a notification object and a userInfo dictionary. The notification object is the NSControl posting the notification. (The field editor of the edited cell originally sends a NSTextDidChangeNotification to the control, which passes it on in this form to its delegate.) The userInfo dictionary contains these keys and values:

Key	Value
@"NSFieldEditor"	The edited cell's field editor

See description of **controlTextDidChange:**, above, for details.

NSControlTextDidEndEditingNotification

This notification object contains a notification object and a userInfo dictionary. The notification object is the NSControl posting the notification. (The field editor of the edited cell originally sends a NSTextDidEndEditingNotification to the control, which passes it on in this form to its delegate.) The userInfo dictionary contains these keys and values:

Key	Value
@"NSFieldEditor"	The edited cell's field editor

See description of **controlTextDidEndEditing:**, above.

NSCursor

Inherits From:	NSObject
Conforms To:	NSCoding NSObject (NSObject)
Declared In:	AppKit/NSCursor.h

Class Description

Instances of the NSCursor class manage the appearance of the cursor. When you initialize a cursor—the designated initializer is **initWithImage:hotSpot:**—you assign it a 16-by-16 pixel NSImage and a point to be the hot spot. The image is usually a small, opaque icon—for example, a pair of cross-hairs—surrounded by transparent pixels. The pixels in the cursor image are mapped on a flipped coordinate system: the upper left pixel is (0,0); the lower right is (15,15).

To determine exactly when the mouse is inside a particular cursor rectangle, the Application Kit tracks a single pixel in the cursor image. This pixel is known as the *hot spot*, and you can reference it using the **hotSpot** method. By definition, the location of the current cursor's hot spot is the location of the mouse; when the hot spot is inside a cursor rectangle, so is the mouse. The hot spot is useful not only for determining which cursor is current, but for determining where a mouse click should have its effect.

An NSCursor object is immutable: you cannot change its hot spot or image after it's created. Instead, use **initWithImage:hotSpot:** to create a new one with the new settings.

An application may use several cursor instances—for example, one that looks like an arrow and one that looks like an I-beam. The instance that currently appears on the screen is called the "current cursor," and is referenced by the **currentCursor** class method. You can set the current cursor in several ways:

- You can send a **set** message to the cursor.
- You can manage cursors in a stack, using the **push** and **pop** methods of NSCursor. The stack's top cursor is the current cursor.
- You can tell a cursor to become current when the mouse enters a part of the screen known as the cursor rectangle.
- You can tell a cursor to set itself when the mouse exits a view's cursor rectangle,

The *cursor rectangle* is a region inside an NSView that triggers a change in the current cursor. To create a cursor rectangle, use the **addCursorRect:cursor:** method of NSView to associate a region of the view with the cursor, as shown in the following example:

```
[aView addCursorRect:aRect cursor:aCursor];
```

```
[aCursor setOnMouseEntered:YES];
```

This assignment means that when the mouse enters *aRect*, *aCursor* will receive a **mouseEntered:** event message, which the cursor uses to make itself the current cursor. However, before the cursor can acknowledge the **mouseEntered:** message, you must invoke the cursor's **setOnMouseEntered:** method. Alternatively, you can set the cursor when the mouse leaves the cursor rectangle by invoking the **setOnMouseExited:** method instead of **setOnMouseEntered:**. A cursor that sets itself upon leaving the cursor rectangle receives a **mouseExited:** event message to instigate the change.

The Application Kit provides two ready-made cursors for commonly used cursor images. You can retrieve these cursors by using the **arrowCursor** and **IBeamCursor** class methods. There is no NSCursor instance for the wait cursor, because the system automatically displays it at the appropriate times.

Adopted Protocols

NSCoding

– encodeWithCoder:
– initWithCoder:

Method Types

Initializing a new cursor

– initWithImage:hotSpot:
– initWithImage:foregroundColorHint:backgroundColorHint: hotSpot:

Setting cursor attributes

– image
– hotSpot
+ hide
+ unhide
+ setHiddenUntilMouseMoves:

Controlling which cursor is current

+ pop
– pop
– push
– set
– mouseEntered:
– setOnMouseEntered:
– isSetOnMouseEntered
– mouseExited:
– setOnMouseExited:
– isSetOnMouseExited

Retrieving cursor instances

+ arrowCursor
+ currentCursor
+ IBeamCursor

Class Methods

arrowCursor

+ (NSCursor *)**arrowCursor**

Returns the default cursor, a slanted arrow with its hot spot at the tip. The arrow cursor is the one you're used to seeing over buttons, scrollers and many other objects in the window system.

See also: + **IBeamCursor**, + **currentCursor**, – **hotSpot**

currentCursor

+ (NSCursor *)**currentCursor**

Returns the cursor that's currently displayed on the screen.

See also: – **set**, – **push**, + **pop**, – **mouseEntered:**, – **mouseExited:**

hide

+ (void)**hide**

Makes the current cursor invisible. If another cursor becomes current, that cursor will be invisible, too. It will remain invisible until you invoke the **unhide** method.

hide overrides **setHiddenUntilMouseMoves:**.

IBeamCursor

+ (NSCursor *)**IBeamCursor**

Returns a cursor that looks like a capital I with a tiny crossbeam at its middle. This is the cursor that you're used to seeing over editable or selectable text. The I-beam cursor's default hot spot is where the crossbeam intersects the I.

See also: + **arrowCursor,** + **currentCursor**

pop

+ (void)**pop**

Pops the current cursor off the top of the stack. The new object on the top of the stack becomes the current cursor. If the current cursor is the only cursor on the stack, this method does nothing.

See also: **– push**

setHiddenUntilMouseMoves:

+ (void)**setHiddenUntilMouseMoves:**(BOOL)*flag*

If *flag* is YES, hides the cursor. The cursor remains invisible until either:

- the mouse moves, or
- you invoke the method again, with *flag* set to NO.

Do not try to counter this method by invoking **unhide**. The results are undefined.

See also: + **hide**

unhide

+ (void)**unhide**

Negates an earlier call to **hide** by showing the current cursor.

See also: + **setHiddenUntilMouseMoves:**, + **hide**

Instance Methods

hotSpot

– (NSPoint)**hotSpot**

Returns the position of the hot spot, specified according to the cursor's flipped 16-by-16 coordinate system. For a more complete explanation, see the class description.

Note that an NSCursor object is immutable: you cannot change its hot spot after it's created. Instead, use **initWithImage:hotSpot:** to create a new one with the new settings.

See also: – **initWithImage:hotSpot:**

image

– (NSImage *)**image**

Returns the image for the receiving cursor, or **nil** if none exists.

Note that an NSCursor object is immutable: you cannot change its hot spot after it's created. Instead, use **initWithImage:hotSpot:** to create a new one with the new settings.

See also: – **initWithImage:hotSpot:**

initWithImage:foregroundColorHint:backgroundColorHint:hotSpot:

– (id)**initWithImage:**(NSImage *)*newImage*
foregroundColorHint:(NSColor *)*fg*
backgroundColorHint:(NSColor *)*bg*
hotSpot:(NSPoint)*hotSpot*

Initializes the receiver, assigns it *anImage* (which must be 16-by-16 pixels) and sets its hot spot to *aPoint*. The foreground and background colors are currently ignored. Returns **self**.

See also: **– initWithImage:hotSpot:**

initWithImage:hotSpot:

– (id)**initWithImage:**(NSImage *)*newImage* **hotSpot:**(NSPoint)*aPoint*

This method is the designated initializer for the class. It initializes the receiver, assigns it *anImage* (which must be 16-by-16 pixels) and sets its hot spot to *aPoint*. Returns **self**.

See also: – **hotSpot**, – **image**, **– initWithImage:foregroundColorHint:backgroundColorHint:hotSpot:**

isSetOnMouseEntered

– (BOOL)**isSetOnMouseEntered**

Returns YES if the receiving cursor will become current when it receives a **mouseEntered:** message; otherwise, returns NO.

To receive such a message, the receiver must first be assigned a cursor rectangle. This assignment can be made using NSView's **addCursorRect:cursor:** method. For a more complete explanation, see the class description.

See also: – **setOnMouseEntered:**, – **isSetOnMouseExited**

isSetOnMouseExited

– (BOOL)**isSetOnMouseExited**

Returns YES if the receiving cursor becomes current when it receives a **mouseExited:** message; otherwise, returns NO.

To receive such a message, the receiver must first be assigned a cursor rectangle. This assignment can be made using NSView's **addCursorRect:cursor:** method. For a more complete explanation, see the class description.

See also: – **setOnMouseExited:**

mouseEntered:

– (void)**mouseEntered:**(NSEvent *)*anEvent*

This message is automatically sent to the receiver when the mouse enters the receiver's cursor rectangle. If used after **setOnMouseEntered:**YES, **mouseEntered:** can make the receiver the current cursor.

In your programs, you won't invoke **mouseEntered:** explicitly. It's only included in the class interface so you can override it.

For a more complete explanation, see the class description.

See also: – **isSetOnMouseEntered, – mouseExited:**

mouseExited:

– (void)**mouseExited:**(NSEvent *)*theEvent*

This message is automatically sent to the receiver when the mouse exits the receiver's cursor rectangle. Like **mouseEntered:**, it is part of the class interface only so you can override it.

See also: – **setOnMouseExited:**, – **isSetOnMouseExited**

pop

– (void)**pop**

Sends a **pop** message to the instance's class.

See also: – **push**, **+ pop**

push

– (void)**push**

Puts the receiver on top of the cursor stack and makes it the current cursor.

See also: – **pop**, **+ pop**

set

– (void)**set**

Makes the receiver the current cursor.

See also: + **currentCursor**

setOnMouseEntered:

– (void)**setOnMouseEntered:**(BOOL)*flag*

If *flag* is YES, the cursor accepts future **mouseEntered:** event messages, otherwise it ignores them. Accepting **mouseEntered:** event messages allows the cursor to be made the current cursor when the mouse enters a view's cursor rectangle.

See also: **– mouseEntered:**

setOnMouseExited:

– (void)**setOnMouseExited:**(BOOL)*flag*

If *flag* is YES, the cursor accepts future **mouseExited:** event messages, otherwise it ignores them. Accepting **mouseExited:** event messages allows the cursor to be made the current cursor when the mouse exits a view's cursor rectangle.

NSCustomImageRep

Inherits From: NSImageRep : NSObject

Conforms To: NSCoding (from NSImageRep)
NSCopying (from NSImageRep)
NSObject (from NSObject)

Declared In: AppKit/NSCustomImageRep.h

Class Description

An NSCustomImageRep is an object that uses a delegated method to render an image. When called upon to produce the image, it sends a message to its delegate to have the method performed.

Like most other kinds of NSImageReps, an NSCustomImageRep is generally used indirectly, through an NSImage object. An NSImage must be able to choose between various representations of a given image. It also needs to provide an off-screen cache of the appropriate depth for any image it uses. It determines this information by querying its NSImageReps.

Thus to work with an NSImage, an NSCustomImageRep must be able to provide some information about its image. Use the following methods, inherited from the NSImageRep class, to set attributes of the NSCustomImageRep:

setSize:

setColorSpaceName:

setAlpha:

setPixelsHigh:

setPixelsWide:

setBitsPerSample:

Note that if these attributes aren't set, and an NSCustomImageRep is used in an NSImage with other representations, NSImage won't be able to select between them. In actual practice, this usually isn't a problem.

Method Types

Initializing a new NSCustomImageRep

– initWithDrawSelector:delegate:

Identifying the object

– delegate
– drawSelector

Instance Methods

delegate

– (id)**delegate**

Returns the delegate object that renders the image for the NSCustomImageRep.

drawSelector

– (SEL)**drawSelector**

Returns the associated draw method selector.

initWithDrawSelector:delegate:

– (id)**initWithDrawSelector:**(SEL)*aMethod* **delegate:**(id)*anObject*

Initializes the receiver, a newly allocated NSCustomImageRep instance, so that it delegates responsibility for rendering the image to *anObject*. When the NSCustomImageRep receives a **draw** message, it will in turn send a message to *anObject* to perform the *aMethod* method. The *aMethod* method should take only one argument, the **id** of the NSCustomImageRep. It should draw the image at location (0.0, 0.0) in the current coordinate system.

Returns **self**.

See also: **– draw** (NSImageRep)

NSDocument

Inherits From:	NSObject
Conforms To:	NSObject (NSObject)
Declared In:	AppKit/NSDocument.h

Class at a Glance

Purpose

NSDocument. is an abstract class that defines the interface for documents, objects that can internally represent data displayed in windows and that can read data from and write data to files. Documents create and manage one

or more window controllers, and are in turn managed by a document controller. Documents respond to first-responder action messages to save, revert, and print their data.

Principal Attributes

- window controllers
- filename
- document type
- print information

Creation

– init:	Designated initializer for new documents
– initWithContentsOfFile:ofType:	For existing documents

Related Classes

NSDocumentController
NSWindowController

Commonly Used Methods

– dataRepresentationOfType:	Returns the document's data in a specified type.
– loadDataReprentation:ofType:	Loads data of a certain type into the document.
– writeToFile:ofType:	Writes the document's data to a file.
– readFromFile:ofType:	Reads the document's data from a file.
– windowNibName	Returns the name of the document's sole nib file (resulting in the creation of a window controller for the window in that file).
- makeWindowControllers	Creates and returns the window controllers used to manage document windows.

Class Description

NSDocument is an abstract class that defines the interface for documents. In a functional sense, a document is a repeatable container for a unique body of information identified by a name under which it is stored. In the context of the Application Kit, a document is an instance of an NSDocument subclass that knows how to represent internally, in one or more formats, the persistent data displayed in windows. A document can read that data from files and write it to files. It is also the first-responder target for many menu commands related to documents, such as Save Document, Revert Document, and Print Document. (When going up the

responder chain, the Application Kit queries a window's NSDocument, if it exists, just after it queries the window delegate, if that is different from the NSDocument.) A document manages its window's edited status and is set up to perform undo and redo operations. When a window is closing, the document is asked before the window delegate to approve the closing.

To create a useful NSDocument subclass you must override some primitive methods and might want to override others. The NSDocument class itself knows how to handle document data as undifferentiated "lumps"; although it understands that these lumps are typed, it knows nothing about particular types. In their overrides of the data-based primitive methods, subclasses must add the knowledge of particular types and how data of the document's native type is structured internally and represented in document windows. Subclasses are also responsible for the creation of the window controllers that manage document windows, and for the implementation of undo and redo. NSDocument takes care of much of the rest, including running Open and Save panels, and generally managing the state of the document. See "“Creating a Subclass of NSDocument”" for more on creating subclasses of NSDocument, particularly the required and optional overridden primitive methods.

NSDocument is one of the triad of Application Kit classes that establish an architectural basis for document-based applications (the others being NSDocumentController and NSWindowController). Read the following section, "“Document-Based Application Architecture”," for the concepts behind this architecture. Following that, "“Implementing a Document-Based Application”" gives the procedure for implementing it.

Document-Based Application Architecture

A document-based application is one of the more common types of applications developed today. They provide a framework for generating identically contained but uniquely composed sets of data that can be stored in files. Word processors and spreadsheet applications are two well-known examples of document-based applications. Before investigating how document-based applications are structured, let's consider exactly what such an application does. It:

- Creates new documents
- Opens existing documents that are stored in files
- Saves documents under user-designated names and locations
- Reverts to saved documents
- Closes documents (usually after prompting the user to save edited documents)
- Prints documents and allows the page layout to be modified
- Represents data of different types internally
- Monitors and sets the document's edited status and validates menu items
- Manages document windows, including setting the window titles
- Handles application and window delegation methods (such as when the application terminates)

Three Application Kit classes provide an architecture for document-based application that simplifies the work developers have to do to implement the features listed above. These classes are NSDocumentController, NSDocument, and NSWindowController.

Objects of these classes divide and orchestrate the work of creating, saving, opening, and managing the documents of an application. They are in tiered one-to-many relationship, as depicted in Figure 1. An application can have only one NSDocumentController, which creates and manages potentially many NSDocument objects (one for each New or Open operation). In turn, an NSDocument object creates and manages one or more NSWindowController objects, one for each of the windows displayed for a document. In addition, some of these objects have responsibilities analogous to NSApplication and NSWindow delegates.

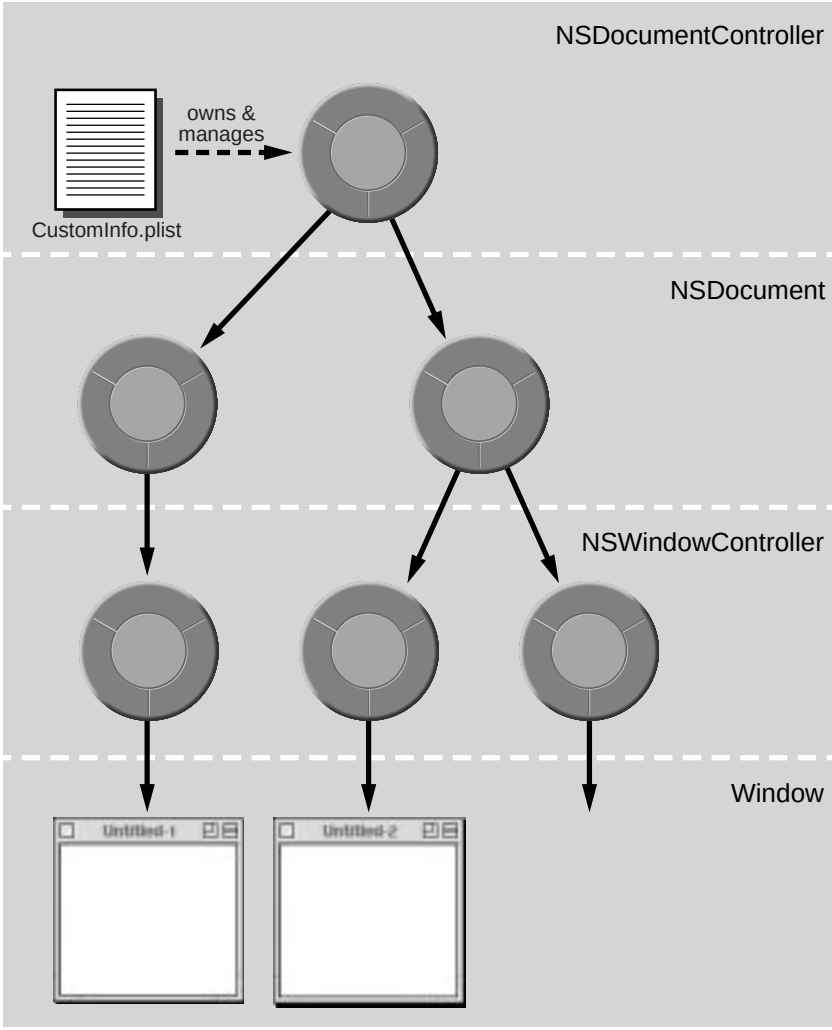


Figure 1 Relationships between NSDocument, NSDocumentController, and NSWindowController

How do these objects divide the work among themselves? The place to start this discussion is not with any of the classes, but with a property list that specifies important details about the application's documents.

The Document Types Info Property List

Each document-based application must include an NSTypes property in its custom info property list (**CustomInfo.plist).** This property specifies information about data types supported by its documents. When the NSDocumentController object creates a new document or opens an existing document, it searches this property list for such items as document class, file extension, whether the type is native or external, and whether external types can be only read or written.

Note: Developers must hand-craft this property list for the current release. In future releases, Project Builder will assist in the creation of this property list.

The following code shows a sample **CustomInfo.plist**; the part that is specific to documents is the NSType property:

```
{
    NSInfoPlistVersion = "5.0";
    NSAppVersion = "1.0";
    NSHumanReadableShortName = "SimpleTextEdit";
    NSHumanReadableCompleteName = "Simple Text Edit Sample";
    NSHumanReadableCopyright = "Copyright (c) 1998, Apple Computer, Inc.";
    NSTypes = (
     {
        NSName = "rtf";
        NSHumanReadableName = "RTF Document";
        NSUnixExtensions = ("rtf");
        NSDOSExtensions = ("rtf");
        NSRole = Editor;
        NSDocumentClass = Document;
      }
    );
}
```

This property list tells the NSDocumentController for the SimpleTextEdit application that the application has one native type ("rtf"), meaning a type of document that, in its role as Editor, it can read and write. The human-readable type name is shown in a pop-up list in the Save Panel's accessory view (but only if there are multiple types). The extensions are used to filter the files shown in Open and Save panels on various platforms; the first extension in each list is automatically added to file names specified in the Save panel. Most important is the document class; NSDocumentController uses this to create an instance of the NSDocument subclass appropriate to a data type. You never have to allocate and initialize your NSDocument explictly in your code; it is done for you.

NSType is a dictionary with key/value pairs defined as follows:

NSName

The abstract name of this data type. This name is used in pasteboard operations. It must be present for the type to be valid, and it must be unique.

NSHumanReadableName

The name of the document type as it appears in the human interface. This name can be localized; the name that appears here is used as the key to extract the localized name from the **InfoPlist.strings** file.

NSIcon

The name of the image file for the icon to represent this type in the human interface. This image file is assumed to reside in the application's Resources directory. It cannot be localized.

NSUnixExtensions

An array of file suffixes used to encode this type on UNIX file systems. The items in the array are interpreted in a case-insensitive manner. The first extension is the one used for newly created files.

NSDOSExtensions

An array of file suffixes used to encode this type on DOS file systems. The array items in the array are interpreted in a case-insensitive manner. The first extension is the one used for newly created files.

NSMacOSTypes

An array of 4-byte MacOS codes used to encode this type. The first extension is the one used for newly created files.

NSMIMETypes

An array of the MIME types used to encode this type.

NSRole

A string that indicates the role of the application for documents of this type: "Editor" (reads, manipulates, displays, and writes), "Viewer" (reads and displays), or "None" (cannot read or write, but is declaring information about the type, such as an icon). If this key is absent, "Editor" is assumed.

NSDocumentClass

The name of the NSDocument subclass used at runtime for loading this document type.

The Role of NSDocumentController

The primary job of an application's NSDocumentController object is to create and open documents, and to track and manage these documents. When a user choses New from the File menu, an NSDocumentController gets the appropriate NSDocument subclass from the NSTypes property, allocates

an instance of this class, and initializes this instance by invoking NSDocument's **init** method. When the user chooses Open from the File menu, NSDocumentController displays the Open panel, gets the user's selection, finds the NSDocument subclass for the file (based on its extension), allocates an instance of this class, and initializes the object and loads document data by invoking NSDocument's **initWithContentsOfFile:ofType:**. In both cases, the NSDocumentController adds a reference to the document object to an internal list to facilitate the management of its documents. It has a notion of the current document as the document whose window is currently key.

NSDocumentController is hard-wired to respond appropriately to certain application events, such as when the application starts up, when it terminates, when the system powers off, and when documents are opened or printed from the Workspace. If you wish, you can make a custom object the application delegate and implement the delegate methods invoked as a result of the same events, and these methods will be invoked instead. However, the default NSDocumentController object is an adequate application controller for most situations, and you should not need to subclass it. If you require additional behavior, such as displaying About panels and handling application preferences, it is recommended that a custom controller object perform these duties rather than a subclass of NSDocumentController.

The Role of NSDocument

The primary job of an NSDocument object is to represent, manipulate, store, and load the persistent data associated with a document. Based on the document types it claims to understand (as specified in the NSTypes property of the info property list), a document must be prepared to:

- Provide the data displayed in windows (and represented internally) in a supported document type.
- Given data of a supported type, load it into internal data structures and display it in windows.
- Store document data in a file at a specified location in the file system.
- Read document data stored in a file.

With the assistance of its window controllers, an NSDocument manages the display and capture of the data in its windows. By some special hard-wiring of the Application Kit, the NSDocument associated with the key window is the recipient of first-responder action messages when users save, print, revert, and close documents. In response to the appropriate action, it knows how to run and manage the Save panel and the Page Layout panel.

A fully implemented NSDocument knows how to track its edited status, print document data, and perform undo and redo operations. Although these behaviors aren't completely provided by default, NSDocument does assist the developer in implementing each. For edited-status tracking, NSDocument provides API for updating a change counter. For undo/redo operations, NSDocument by default lazily creates an NSUndoManager when one is requested, responds appropriately to Undo and Redo menu commands, and updates the change counter when undo and redo operations are performed. For printing, NSDocument facilitates the display of the Page Layout panel and the subsequent modification of the NSPrintInfo object used in printing.

Creating a Subclass of NSDocument

Every application that takes advantage of the Application Kit's architecture for document-based applications must create at least one subclass of NSDocument. This architecture requires that you override some NSDocument methods in an either/or scenario, and recommends overriding several others in certain situations.

- **Data-based primitives**. The **dataRepresentationOfType:** method has to be implemented to create and return document data (packaged as an NSData object) of a supported type, usually in preparation for writing that data to a file. The **loadDataRepresentation:ofType:** method must be implemented to convert an NSData object containing document data of a certain type into the document's internal data structures and display that data in a document window; the NSData object usually results from the document reading a document file. Subclasses must override these methods.
- **Location-based primitives**. By default the **writeToFile:ofType:** method writes data to a file after obtaining the data from the **fileWrapperRepresentationOfType:**, which gets it from the **dataRepresentationOfType:** method. The **readFromFile:ofType:** method reads data from a file, creates an NSFileWrapper object from it, and gives this object to **loadFileWrapperRepresentation: ofType:**; if this object represents a simple file, it is passed to the **loadDataRepresentation:ofType:** method for processing; otherwise (that is, the object represents a directory), the **loadFileWrapperRepresentation:ofType:** method is overriden to handle the situation. Subclasses can override any of these methods instead of the data-based primitives if the way NSDocument reads and writes document data is not sufficient; their override implementations, however, must also assume the loading duties of the data-based primitives.
- **Window controller creation**. NSDocument subclasses must also create their window controllers. They can do this indirectly or directly. If a document has only one nib file (with one window in it), the subclass can override **windowNibName** to return the name of the window nib file; as a consequence, a default NSWindowController instance is created for the document, with the document as the nib file's owner. If a document has multiple windows, or if an instance of a custom NSWindowController subclass is to be used, the NSDocument subclass must override **makeWindowControllers** to create these objects.
- **Printing and page layout**. Normally, a document-based application can change the information it uses to define how document data is printed (an NSPrintInfo object). Subclasses can override **shouldChangePrintInfo:** to disallow this change. If an application is to print document data, subclasses of NSDocument must override **printShowingPrintPanel:**.
- **Backup files**. When it saves a document, NSDocument creates a backup of the old file before it writes data to the new one (backup files have the same name as the new file, but with a tilde just before the extension). Normally, if the write operation is successful, it deletes the backup file. Subclasses can override **keepBackupFile** to return YES, and thus retain the most recent backup file.
- **Save panel accessory view**. By default, when NSDocument runs the Save panel, and the document has multiple writable document types, it inserts an accessory view near the bottom of the panel. This view contains a pop-up list of the writable types. If you don't want this pop-up list, override **shouldRunSavePanelWithAccessoryView** to return NO.

- **Menu items**. NSDocument implements **validateMenuItem:** to manage the enabled state of the Revert and Save As menu items. If you want to validate other menu items, you can override this method, but be sure to invoke **super**'s implementation. For more information on menu item validation, see the description of the NSMenuValidation informal protocol.

The initializers of NSDocument are another issue for subclassers. The **init** method is the primary initializer, and it is invoked by the other initializer **initWithContentsOfFile:ofType:**. The **init** method is directly invoked when a new document is created; the **initWithContentsOfFile:ofType:** method is directly invoked when a document is opened. Therefore if you have any initializations that apply only to documents that are opened, you should override **initWithContentsOfFile:ofType:**; if you have general initializations, you should, of course, override **init**. In both cases, be sure to invoke **super**'s implementation as the first thing.

The Role of NSWindowController

An NSWindowController manages one window associated with a document, which is usually stored in a nib file. If a document had multiple windows, each window would have its own window controller. For example, a document might have a main data-entry window and a window listing records for selection; each window would have its own NSWindow Controller. When requested by its owning NSDocument, an NSWindowController loads the nib file containing a window and displays it. It also assumes responsibility for properly closing windows (after ensuring that they are saved).

The NSWindowController offers additional behavior to document-based applications. It can store the size and location of windows in the user defaults database (this may not be desired behavior for most document-based applications, because it could quickly overpopulate the defaults database with document window-frame entries). It also cascades document windows in relation to each other, so they don't completely obstruct one another.

Subclasses of NSWindowController are optional. Applications can often use the default instance. Subclasses can augment NSWindowControllers to perform different nib-loading and setup tasks or to customize the titles of windows.

Implementing a Document-Based Application

It is possible to put together a document-based application without having to write much code. If your requirements are minimal, the Application Kit provides you with a default NSWindowController instance and a default NSDocumentController instance. You just have to create a document project, compose the human interface, implement a subclass of NSDocument, and add any other custom classes or behavior required by your application.

The following procedures step you through the tasks you must do, and might want to do, when implementing a document-based application. Where something is described in detail elsewhere, such as overridden methods in NSDocument subclasses, you are referred there.

As for the three classes behind document-based applications, two likely concerns are the number of required objects and whether subclassing is necessary. The following table summarizes this information:

Class	How Many Objects?	Subclass?
NSDocumentController	1 per application	Optional (but unlikely)
NSDocument	1 per saved file	Required
NSWindowController	1 per document nib file	Optional (but likely)

The Document-Based Application Package

The Yellow Box development environment provides a Document-Based Application project type to expedite the development of these kinds of applications. This project type provides the following things:

- **A nib file for the application's document**. A subclass of NSDocument named "Document" is made File's Owner of the nib file. It has an outlet to its window. The window is "blank" (that is, without object on it).
- **The application's main nib file**. This nib file contains an application menu with a File menu (with all of its associated document commands) and Undo and Redo menu items in the Edit menu. These menu items, as well as all of the menu items of the File menu, are connected to the appropriate first-responder action methods. The About MyApp menu item of the Apple (or Info) menu is connected to the action method for loading the nib file containing the About panel.
- **A nib file for the About panel**. This nib file contains a conventional About dialog with placeholder text for application name, version string, and so on. The custom NSWindowController object (see below) is File's Owner of this nib file.
- **A skeletal NSDocument subclass implementation**. The project includes **Document.h** and **Document.m**, which are derived from the definition of the NSDocument subclass in the document nib file. The latter file includes empty but commented blocks for the **dataRepresentationOfType:**, **loadDataRepresentation:ofType:**, and **windowControllerDidLoadNib:** methods. It also includes a fully implemented **windowNibName** method.
- **A custom NSWindowController subclass for the About panel**.
- **A custom info property list template**. In the Other Resources category of the project is a **CustomInfo.plist** file that contains placeholder values for global application keys as well as NSTypes keys.

The following procedure describes what you must do to create a document-based application when you use the project type developed for it. If this project type is absent on your system, you will have to complete the tasks listed above yourself. For this eventuality, the following table lists the appropriate first-responder

action connections to make (that is, in Interface Builder connect the menu command to the "1" icon in the nib file window):

File Menu Command	First-Responder Action
New	newDocument:
Open	openDocument:
Save	saveDocument:
Save As	saveAsDocument:
Save To	saveToDocument:
Save All	saveAllDocuments:
Close	closeDocument:
Revert	revertDocument:
Print	printDocument:
Page Layout	runPageLayout:

Also, if you have added Undo and Redo menu items in the Edit menu, connect them to the first-responder **undo:** and **redo:** methods.

Create the Project and Compose the Interface

1. Launch Project Builder and choose New from the Project.

   In the New Project panel, see if the pop-up menu of project types includes "Document Based Application." If it doesn't, you will have to load this project type, located in **/System/Developer/ProjectTypes** (on Yellow Box for Windows platforms, prepend the value of NEXT_ROOT to this path).

2. Create a project of type Document Based Application..

3. Double-click the **Document.nib** file in Project Builder's Interfaces category to open this file in Interface Builder.

   If you want the nib file to be named something else, you can save it under another name in Interface Builder and add it to the project. If you do this, you must also modify the string returned by the **windowNibName** method in the NSDocument subclass implementation.

4. Create the human interface of the document window.

5. If the objects on the document window require further outlets or actions, add these to the Document subclass of NSDocument. Connect these actions and outlets via the File's Owner icon on the Instances display of the nib file window.

Warning: Do not generate an instance of Document to make these connections.

If you want your NSDocument subclass named something other than "Document," change the name in Interface Builder and wherever it occurs in the Document header and implementation (**.m**) files.

6. If your document objects interacts with other custom objects, such as model objects that perform specialized computations, define those objects in Interface Builder and make any necessary connections to them.

Complete the Custom Info Property List

1. In Project Builder, select **CustomInfo.plist** in the Other Resources category.
2. Replace the placeholder or default values in the NSType property list with those specific to your document. If your document has more than one document type, duplicate the given NSType property and fill it in with the appropriate values.

   See "“The Document Types Info Property List”" for information on the key/value pairs specific to documents.

3. For the global application properties enter your application's names, version, and copyright information.

Implement the NSDocument Subclass

The following procedure just gives general guidelines. For details see the complete NSDocument specification and especially "“Creating a Subclass of NSDocument”." You might also want to read the class specifications of NSUndoManager, NSPasteboard, and the print classes.

1. In Project Builder, open the header file of your NSDocument subclass located in the Headers category.
2. If you added outlets or actions to your NSDocument subclass in Inteface Builder, add them to the subclass' header file. Also any other required instance variables and include the declarations of new methods that you wish to be public, such as accessor methods.

   Of course, you can specify additional outlets in actions in the existing header file and then import them into the nib file by using Interface Builder's Classes>Read File command.

3. Open the subclass implementation file (**.m**) in Project Builder's Classes category.
4. Although it's not usually necessary, you can override the primary initializer (**init**) and perhaps the document-opening initializer **initWithContentsOfFile:ofType:** to perform initializations specific to you subclass; be sure to invoke **super**'s implementations. You can also implement **awakeFromNib** to initialize objects unarchived from the document's window nib files.
5. Override the data-based primitives (null implementations of these methods provided by the project type).

In almost all cases, you should implement **dataRepresentationOfType:** (to provide document data of a certain type) and **loadDataRepresentation:ofType:** (to load document data of a certain type).

Example:

```
// ---------------------------------------------------------------------------

- (NSData *)dataRepresentationOfType:(NSString *)aType {
    NSAssert([aType isEqualToString:@"rtf"], @"Unknown type");
    return [textView RTFFromRange:NSMakeRange(0, [[textView textStorage] length])];
}

// ---------------------------------------------------------------------------

- (BOOL)loadDataRepresentation:(NSData *)data ofType:(NSString *)aType {
    NSAssert([aType isEqualToString:@"rtf"], @"Unknown type");
    fileContents = [data copyWithZone:[self zone]];
    return YES;
}
```

6. If you need to read or write document data in a special way (because, for example, document data is stored in a file package), you can override **readFromFile:ofType:** and **writeToFile:ofType:** to not only read and write data, but load and provide data in a given type.

   Example:

```
// ---------------------------------------------------------------------------

- (BOOL)writeToFile:(NSString *)fileName ofType:(NSString *)type {
    return [[textView string] writeToFile:fileName atomically:YES];
}

// ---------------------------------------------------------------------------

- (BOOL)readFromFile:(NSString *)fileName ofType:(NSString *)type {
    fileContents = [[NSString alloc] initWithContentsOfFile:fileName];
    return fileContents != nil;
}
```

7. Create the window controllers for the NSDocument object.

   If your document has only window, the project types provides a default implementation :

```
- (NSString *)windowNibName {
    return @"Document";
}
```

   If your document has more than one window, or if you have a custom subclass of NSWindowController, override **makeWindowControllers**. Make sure you add each created window controller to the list of such objects managed by the document (**addWindowController:**).

8. You can implement **windowControllerWillLoadNib:** and **windowControllerDidLoadNib:** to perform any necessary tasks related to the window before and after it is loaded from the nib file.

   Example:

```
- (void)windowControllerDidLoadWindowNib:(NSWindowController *)windowController {
    [super windowControllerDidLoadWindowNib:windowController];
    [textView setAllowsUndo:YES];
    if (fileContents != nil) {
        [textView setString:fileContents];
        [fileContents release];// Don't need it anymore
        fileContents = nil;
    }
}
```

9. Mark the document's "dirty" flag when it is edited.

   The flag returned by **isDocumentEdited** indicates whether the document has unsaved changes. Although NSDocument clears this flag when it saves or reverts a document, you must set this flag in your code, unless you are using NSDocument's default undo/redo mechanism. Normally, you respond to the appropriate delegation or notification messages sent when users edit a document, then invoke **updateChangeCount:** with an argument of NSChangeDone to set the "dirty" flag.

10. Write the code that prints the document's data.

    If you want users to be able to print a document, you must override **printShowingPrintPanel:**, possibly providing a modified NSPrintInfo object.

11. Register undo and redo groups in your code. See the class description of NSUndoManager for details.

And of course, you implement any methods that are special to your NSDocument subclass.

Implement Additional Controller Classes

If the default NSWindowController instance provided by the application kit does not meet the needs of your document-based application, you can create a custom subclass of it. If you do so, you must override NSDocument's **makeWindowControllers** to instantiate this custom class and add the created object to the document's list of window controllers.

If the default NSDocumentController object somehow does not meet all of your requirements for an application controller, such as handling user preferences or responding to uncommon application delegate messages, you should create a separate controller object (instead of subclassing NSDocumentController). For information on implementing NSDocumentController and NSWindowController subclasses, refer to the appropriate class specifications.

Method Types

Initializing an NSDocument

– init
– initWithContentsOfFile:ofType:

Loading and representing document data

– dataRepresentationOfType:
– loadDataRepresentation:ofType:
– fileWrapperRepresentationOfType:
– loadFileWrapperRepresentation:ofType:

Creating and managing window controllers

– makeWindowControllers
– windowNibName
– windowControllerDidLoadNib:
– windowControllerWillLoadNib:
– windowControllers
– addWindowController:
– shouldCloseWindowController:

Showing document windows

– showWindows
– displayName

Reading from and writing to files

– readFromFile:ofType:
– writeToFile:ofType:
– fileNameFromRunningSavePanelForSaveOperation:
– fileName
– setFileName:
– runModalSavePanel:withAccessoryView:
– shouldRunSavePanelWithAccessoryView
– keepBackupFile

Managing document edited status

– isDocumentEdited
– updateChangeCount:

Responding to user actions

– closeDocument:
– printDocument:
– runPageLayout:
– revertDocumentToSaved:
– saveDocument:
– saveDocumentAs:
– saveDocumentTo:

Closing documents	
	– canCloseDocument
	– close
Reverting documents	
	– revertToSavedFromFile:ofType:
Printing documents	
	– printShowingPrintPanel:
	– printInfo
	– setPrintInfo:
	– runModalPageLayoutWithPrintInfo:
	– shouldChangePrintInfo:
Managing file types	
	– setFileType:
	– fileType
	+ isNativeType:
	+ readableTypes
	+ writableTypes
Managing menu commands	
	– validateMenuItem:

Class Methods

isNativeType:

+ (BOOL)**isNativeType:**(NSString *)*aType*

Returns whether document data of type *aType* is a native type, that is, one that a document can both read and write.

See also: + **readableTypes** , + **writableTypes**

readableTypes

+ (NSArray *)**readableTypes**

Returns the types of data that a document can read natively and any types filterable to that native type.

See also: + **isNativeType:**, + **writableTypes**

writableTypes

+ (NSArray *)**writableTypes**

Returns the types of data that a document can write natively and any types filterable to that native type.

See also: + **isNativeType:**, + **readableTypes**

Instance Methods

addWindowController:

– (void)**addWindowController:**(NSWindowController *)*aController*

Adds the window controller *aController* to the list of window controllers associated with the receiver. An NSDocument uses this list when it displays all document windows, sets window edited status upon a undo or redo operation, and modifies window titles. The method also sets the document outlet of the window controller to **self** if it is not already set. If you create window controllers by overriding **windowNibName**, this method is invoked automatically. If you create window controllers in **makeWindowControllers** or in any other context, such as in response to a user event, you should invoke this method for each created window controller. To remove a window controller from the list of active controllers, send it an (NSWindowController) **close** message.

See also: – **setDocument:** (NSWindowController)

canCloseDocument

– (BOOL)**canCloseDocument**

Returns whether the receiver can be closed. If the document has unsaved changes, the method displays an attention panel asking users if they want to save the document. It returns NO only if the user clicks the Cancel button or if the document attempts to save itself but was somehow unsuccessful. Otherwise—if the document has no unsaved changes, if the document was successfully saved, or if the user clicks Don't Save—it returns YES. In most situations, you should receive YES from this method before closing the document.

See also: – **close**, – **saveDocument:**, – **shouldCloseWindowController:**

close

– (void)**close**

Closes all windows owned by the document and removes the receiver from the list of documents maintained by the document controller, which consequently releases it. This method closes the document immediately, without asking users if they want to save the document.

See also: – **canCloseDocument**, – **closeDocument:**, – **shouldCloseWindowController:**

closeDocument:

– (IBAction)**closeDocument:**(id)*sender*

The action invoked in the receiver when the user choses the Close Document menu command. The target of the action message must be set to **nil** so that the receiver can respond to it as first responder. The default implementation of this method closes the document only if **canCloseDocument** returns YES.

See also: – **close**

dataRepresentationOfType:

– (NSData *)**dataRepresentationOfType:**(NSString *)*aType*

A primitive method overriden by sublcasses to return a data object that represents the data of the receiver in a given type (*aType*). The default implementation raises an NSInternalInconsistencyException. This method is invoked by the default implementation of **fileWrapperRepresentationOfType:**.

Here is a typical implementation:

```
- (NSData *)dataRepresentationOfType:(NSString *)aType {
    NSAssert([aType isEqualToString:@"rtf"], @"Unknown type");
    return [textView RTFFromRange:NSMakeRange(0, [[textView textStorage] length])];
}
```

See also: – **loadDataRepresentation:ofType:**

displayName

– (NSString *)**displayName**

Returns the name of the document as displayed in the title bars of the document's windows and in attention panels related to the document. Returns an empty string if the document is new and hasn't been saved. If the document has been saved, the display name is the last component of the directory location of the saved file (for example, "MyDocument" if the path is "/tmp/MyDocument.rtf"). If the document is new, NSDocument makes the display name "Untitled-*n*" where *n* is a number in a sequence of new and unsaved

documents. Subclasses of NSWindowController can override **windowTitleForDisplayName:** to modify the display name as it appears in window titles.

fileName

– (NSString *)**fileName**

Returns the filename (as a fully qualified path) under which the document has been saved.

See also: – **fileNameFromRunningSavePanelForSaveOperation:**, – **setFileName:**

fileNameFromRunningSavePanelForSaveOperation:

– (NSString *)**fileNameFromRunningSavePanelForSaveOperation:**
(NSSaveOperationType)*saveOperation*

Runs the modal Save panel and returns the filename (as a fully qualified path) selected for the document. The *saveOperation* constant determines the title of the Save panel (Save, Save As, Save To). It also affects whether the Save Panel includes an accessory view with a pop-up list containing the document's native or writable types. If *saveOperation* is NSSaveOperation or NSSaveAsOperation, the accessory pop-up list contains only those document types the application can read and write. If saveOperation is NSSaveToOperation, the pop-up lists additionally includes the document types that the application can write (but can't read). If there is only one type the document can be written to, or if **shouldRunSavePanelWithAccessoryView** returns NO, the accessory view isn't shown. The default extension for saved documents is the first extension assigned for the document's native type or, if there is no native type, the extension for the first writable type specified in the NSTypes property. File packages are treated as files.

See also: – **fileName**, – **runModalSavePanel:withAccessoryView:**

fileType

– (NSString *)**fileType**

Returns the document type under which the document is saved. When a document is saved, the type is determined by the file extension, as defined in the custom info dictionary (specified in **CustomInfo.plist)**.

See also: – **setFileType:**

fileWrapperRepresentationOfType:

– (NSFileWrapper *)**fileWrapperRepresentationOfType:**(NSString *)*aType*

Returns an NSFileWrapper object that represents the data of the receiver in a given type (*aType*). This method invokes **dataRepresentationOfType:** to get the data object from which to create a plain-file file wrapper. Subclasses can override this method if **dataRepresentationOfType:** is not adequate for their needs. This method is invoked by the default implementation of **writeToFile:ofType:**.

See also: – **loadFileWrapperRepresentation:ofType:**

hasUndoManager

– (BOOL)**hasUndoManager**

Returns whether the receiver owns or should own an NSUndoManager.

See also: – **setHasUndoManager:**

init

– (id)**init**

Initializes and returns an NSDocument object. This initializer (the designated initializer) is typically invoked by NSDocumentController's **makeUntitledDocumentOfType:**.

initWithContentsOfFile:ofType:

– (id)**initWithContentsOfFile:**(NSString *)*fileName* **ofType:**(NSString *)*docType*

Initializes and returns an NSDocument object of document type *docType* and containing data stored in the file at the path location *fileName*. If the file cannot be opened, displays an attention panel informing the user, and then returns **nil**. In opening the file, invokes the **readFromFile:ofType:** method. If it successfully opens the file, it "remembers" *fileName* and *docType* (through **setFileName:** and **setFileType:**). This initializer is typically invoked by NSDocumentController's **makeDocumentWithContentsOfFile: ofType:**.

isDocumentEdited

– (BOOL)**isDocumentEdited**

Returns YES if the document has been edited since it was last saved or if the document is new; otherwise, returns NO. The edited status of each document window reflects the document's edited status.

See also: – **updateChangeCount:**, **setDocumentEdited:** (NSWindow)

keepBackupFile

– (BOOL)**keepBackupFile**

Returns whether the receiver should keep the backup files created before document data is written to a file (NO by default). Override this method if you want different behavior.

See also: – **writeToFile:ofType:**

loadDataRepresentation:ofType:

– (BOOL)**loadDataRepresentation:**(NSData *)*docData* **ofType:**(NSString *)*docType*

Overridden by subclasses to load document data (*docData*) of type *docType* into the document, display it in windows, and return whether the operation was successful. This method is typically invoked by **loadFileWrapperRepresentation:ofType:** after an NSData object is created from the contents of the file wrapper (which can include directories). The default implementation raises an NSInternalInconsistencyException and returns NO. Subclasses must override this method unless they override **readFromFile:ofType:** or **loadFileWrapperRepresentation:ofType:** to do specialized reading and loading of document data.

Here is an example implementation:

```
- (BOOL)loadDataRepresentation:(NSData *)data ofType:(NSString *)aType {
    NSAssert([aType isEqualToString:@"rtf"], @"Unknown type");
    fileContents = [data copyWithZone:[self zone]];
    return YES;
}
```

See also: – **dataRepresentationOfType:**

loadFileWrapperRepresentation:ofType:

– (BOOL)**loadFileWrapperRepresentation:**(NSFileWrapper *)*wrapper*
ofType:(NSString *)*docType*

Load document data in file wrapper *wrapper* of type *docType* into the document, displays it in windows, and returns whether the operation was successful. If wrapper is a simple file, it invokes **loadDataRepresentation:ofType:** load the data. If wrapper is a directory, it returns NO by default; subclasses can override to handle file wrappers containing directories. This method is typically invoked by **readFromFile:ofType:** after it creates an NSData object from the contents of the file.

See also: – **fileWrapperRepresentationOfType:**

makeWindowControllers

– (NSArray *)**makeWindowControllers**

Overridden by subclasses to create and return multiple window controllers (NSWindowController objects) or a single window controller derived from a custom subclass of NSWindowController. A document-based application must have one window controller per document nib file (each of which must contain a primary window controlled by the window controller). You usually should add each created window controller to the document's list of such objects by invoking **addWindowController:**.

If subclasses do not override this method, they must override **windowNibName**, but they should do so only if the following two conditions apply:

- The document has a single nib file (with its one window).
- The default NSWindowController instance is sufficient to manage this window.

If you override this method, **windowControllerDidLoadNib:** and **windowControllerWillLoadNib:** are invoked, as they are when **windowNibName** is implemented.

The default implementation raises NSInternalInconsistencyException and returns **nil**.

See also: – **windowControllers**

printDocument:

– (void)**printDocument:**(id)*sender*

Prints the document in response to the user choosing the Print menu command. An NSDocument receives this action message as it travels up the responder chain. The default implementation invokes **printShowingPrintPanel:** with an argument of YES.

See also: – **printInfo**, – **runPageLayout:**, – **setPrintInfo:**, – **shouldChangePrintInfo:**

printInfo

– (NSPrintInfo *)**printInfo**

Returns the document's customized NSPrintInfo object or the default NSPrintInfo instance. The document's copy of the NSPrintInfo object can either be directly set, or set as a result of running the Page Layout panel. A subclass can override this method to always return the shared NSPrintInfo instance if it does not want its own copy.

See also: – **runPageLayout:**, – **setPrintInfo:**, – **shouldChangePrintInfo:**,

printShowingPrintPanel:

– (void)**printShowingPrintPanel:**(BOOL)*flag*

Overridden by subclasses to print the current document's (the receiver's) data; if flag is YES, the implementation should first display the Print panel. This method is typically invoked by **printDocument:** with an argument of YES. The default implementation does nothing. If there is any printing information other than that encoded in the receiver's NSPrintInfo object, sublcasses should get it here.

See also: – **printInfo**

readFromFile:ofType:

– (BOOL)**readFromFile:**(NSString *)*fileName* **ofType:**(NSString *)*docType*

Reads and loads document data of type *docType* from the file at path *fileName*, returning whether the operation was successful. This method invokes **loadDataRepresentation:ofType:** and is invoked when the receiver is first created and initialized by **initWithContentsOfFile:ofType:**. It uses NSData's **initWithContentsOfFile:** to get the document data.

This method is one of the location-based primitives. Subclasses can override this method instead of overriding **loadDataRepresentation:ofType:** to read *and* load document data. Subclasses that handle file packages such as RTFD or that treat locations of files as anything other than paths should override this method. Override implementations of this method can filter the document data using NSPasteboard's or other filtering services.

See also: – **dataRepresentationOfType:**, – **writeToFile:ofType:**

revertDocumentToSaved:

– (void)**revertDocumentToSaved:**(id)*sender*

The action method invoked in the receiver as first responder when the user chooses the Revert menu command. The default implementation displays an attention panel to confirm the user's intentions. If the user confirms the command, the method reverts the document to the data saved in the file system by invoking **revertDocumentToSaved:**. If the operation is successful, it clears the update count..

See also: – **updateChangeCount:**

revertToSavedFromFile:ofType:

– (BOOL)**revertToSavedFromFile:**(NSString *)*fileName* **ofType:**(NSString *)*type*

Reverts the receiver to the data stored in the file system. Invokes **readFromFile:ofType:** and returns whether that method successfully read the file and processed the document data.

See also: – **revertDocumentToSaved:**

runModalPageLayoutWithPrintInfo:

– (int)**runModalPageLayoutWithPrintInfo:**(NSPrintInfo *)*printInfo*

Runs the Page Layout modal panel with the document's printing information object (*printInfo*) as argument and returns the result constant (indicating the key pressed by the user). Invoked by **runPageLayout:**.

See also: – **shouldChangePrintInfo:**, – **runModalWithPrintInfo:** (NSPageLayout)

runModalSavePanel:withAccessoryView:

– (int)**runModalSavePanel:**(NSSavePanel *)*savePanel*
withAccessoryView:(NSView *)*accessoryView*

Runs the modal Save panel *savePanel* with accessory view *accessoryView* and returns the result constant (indicating the button clicked by the user). The accessory view is usually a pop-list containing the document's native types and its supported writable types. Invoked by **fileNameFromRunningSavePanelForSaveOperation:**.

See also: – **shouldRunSavePanelWithAccessoryView**

runPageLayout:

– (void)**runPageLayout:**(id)*sender*

The action method invoked in the receiver as first responder when the user chooses the Page Layout menu command. The default implementation invokes **runModalPageLayoutWithPrintInfo:** with the document's current NSPrintInfo object as argument; if the user presses the OK button, and the document authorizes changes to its printing information (**shouldChangePrintInfo:**), the method sets the document's new NSPrintInfo object and increments the document's change count.

See also: – **setPrintInfo:**, – **updateChangeCount:**

saveDocument:

– (void)**saveDocument:**(id)*sender*

The action method invoked in the receiver as first responder when the user chooses the Save menu command. The default implement saves the document in two different ways, depending on whether the document has a file path and a document type assigned. If path and type are assigned, it simply writes the document under it's current file path and type after making a backup copy of the previous file. If the document is new (no file path and type), it runs the modal Save panel to get the file location under which to save the document. It writes the document to this file, sets the document's file location and document type (if a native type), and clears the document's edited status.

See also: – **fileNameFromRunningSavePanelForSaveOperation:**, – **setFileName:**, – **setFileType:**, – **updateChangeCount:**,

saveDocumentAs:

– (void)**saveDocumentAs:**(id)*sender*

The action method invoked in the receiver as first responder when the user chooses the Save As menu command. The default implementation runs the modal Save panel to get the file location under which to save the document. It writes the document to this file, sets the document's file location and document type (if a native type), and clears the document's edited status.

See also: – **fileNameFromRunningSavePanelForSaveOperation:**, – **setFileName:**, – **setFileType:**, – **updateChangeCount:**

saveDocumentTo:

– (void)**saveDocumentTo:**(id)*sender*

The action method invoked in the receiver as first responder when the user chooses the Save To menu command. The default implementation is identical to **saveDocumentAs:**, except that this method doesn't clear the document's edited status and doesn't reset file location and document type if the document is a native type.

See also: – **fileNameFromRunningSavePanelForSaveOperation:**

setFileName:

– (void)**setFileName:**(NSString *)*fileName*

Sets the file (filename and directory path) under which document data is saved to *fileName*. As a side effect, synchronizes the titles of the document's windows with the new name or location. Under normal

circumstances, the receiver's filename is set when it saved as a new document (Save) or when an existing document is saved under a different filename or path (Save As).

See also: – **fileName**

setFileType:

– (void)**setFileType:**(NSString *)*docType*

Sets the document type under which the file is saved to *docType*. The document type affects how the data is filtered when it is written to or read from a file.

See also: – **fileType**

setHasUndoManager:

– (void)**setHasUndoManager**:(BOOL)*flag*

Sets whether the receiver has its own NSUndoManager. If *flag* is NO and the receiver currently owns an NSUndoManager, the NSUndoManager is released after being removed as observer of undo-related notifications.

See also: – **hasUndoManager**

setPrintInfo:

– (void)**setPrintInfo:**(NSPrintInfo *)*printInfo*

Sets the document's NSPrintInfo object to *printInfo*; this object is used in laying out the document for printing.

See also: – **printInfo**

setUndoManager:

– (void)**setUndoManager:**(NSUndoManager *)*undoManager*

Sets the undo manager owned by the receiver to *undoManager* and releases any undo manager currently owned by the receiver. If *undoManager* is **nil**, it turns off the **hasUndoManager** flag. If *undoManager* is non-**nil**, it adds the receiver as an observer of NSUndoManagerDidUndoChangeNotification, NSUndoManagerDidRedoChangeNotification, and NSUndoManagerWillCloseUndoGroupNotification.

See also: – **undoManager**, NSUndoManager (class)

shouldChangePrintInfo:

– (BOOL)**shouldChangePrintInfo:**(NSPrintInfo *)*newPrintInfo*

Returns whether the receiver should allow changes to the default NSPrintInfo object used in printing the document. The default implementation returns YES. Subclasses can override this method to return NO. This method is invoked by the **runPageLayout:** method, which sets a new NSPrintInfo for the document only if this method returns YES.

shouldCloseWindowController:

– (BOOL)**shouldCloseWindowController:**(NSWindowController *)*windowController*

If closing the *windowController* would cause the receiver to be closed, invokes **canCloseDocument** to display a Save panel and give the user an opportunity to save the document. Returns NO if that is what **canCloseDocument** returns (that is, if the save operation was unsuccessful or the user clicked Cancel in the Save dialog); otherwise returns YES. Note that the receiver doesn't close until its window controller closes.

See also: – **close**, – **shouldCloseDocument** (NSWindowController)

shouldRunSavePanelWithAccessoryView

– (BOOL)**shouldRunSavePanelWithAccessoryView**

Returns YES by default; as a result, when NSDocument displays the Save panel, it includes an accessory view containing a pop-up list of supported writable document types. Subclasses can override to return NO, thus excluding the accessory view from the Save panel.

Here is an example implementation:

```
- (BOOL)shouldRunSavePanelWithAccessoryView {
    return [self fileName] == nil;
}
```

See also: – **runModalSavePanel:withAccessoryView:**

showWindows

– (void)**showWindows**

Displays all windows of the document, bringing them to the front and making them main or key, as necessary.

undoManager

– (NSUndoManager *)**undoManager**

Returns the NSUndoManager used by the document or **nil** if the receiver should not own one. If the undo manager doesn't exist and **hasUndoManager** returns YES, it creates one and invokes **setUndoManager:** with the NSUndoManager as argument.

updateChangeCount:

– (void)**updateChangeCount:**(NSDocumentChangeType)*changeType*

Updates the document's change count according to *changeType*. The change count indicates the document's edited status; if the change count is zero, the document has no changes to save, and if the change count is greater than zero, the document has been edited and is unsaved. The *changeType* constant can increment (NSChangeDone), decrement (NSChangeUndone) , or set to zero (NSChangeClear) the change count. If you are implementing undo and redo in an application, you should increment the change count every time you create an undo group, and decrement the change count when an undo or redo operation is performed.

Note that if you are using NSDocument's default undo/redo features, setting the document's edited status by updating the change count happens automatically. You only need to invoke this method when you are not using these features.

See also: NSUndoManager

validateMenuItem:

– (BOOL)**validateMenuItem:**(NSMenuItem *)*anItem*

Validates the Revert menu item and items selected from the Save panel's pop-up list of writable document types items. Returns YES if *anItem* should be enabled, NO otherwise. Returns YES for Revert if the document has been edited and a file exists for the document. Returns YES for an item representing a writable type if, during a Save or Save As operation, it is a native type for the document. Subclasses can override this method to perform additional validations.

windowControllerDidLoadNib:

– (void)**windowControllerDidLoadNib:**(NSWindowController *)*windowController*

Overridden by subclass to perform any tasks after the document's window controller (*windowController*) loads the nib file containing the document window. The default implementation does nothing.

See also: – **windowControllerWillLoadNib:**, – **windowControllers**

windowControllerWillLoadNib:

– (void)**windowControllerWillLoadNib:**(NSWindowController *)*windowController*

Overridden by subclass to perform any tasks before the document's window controller (*windowController*) loads the nib file containing the document window. The default implementation does nothing.

See also: – **windowControllerDidLoadNib:**, – **windowControllers**

windowControllers

– (NSArray *)**windowControllers**

Returns the document's current window controllers (NSWindowController objects). If there are no window controllers, returns an empty NSArray.

See also: – **makeWindowControllers**, – **windowControllerDidLoadNib:**,
– **windowControllerWillLoadNib:**, – **windowNibName**

windowNibName

– (NSString *)**windowNibName**

Overridden by subclasses to return the name of the document's sole nib file. Using this name, NSDocument creates and instantiates a default instance of NSWindowController to manage the window. If your document has multiple nib files, each with its own single window, or if the default NSWindowController instance is not adequate for your purposes, you should override **makeWindowControllers**.

The default implementation returns **nil**.

See also: – **windowControllers**

writeToFile:ofType:

– (BOOL)**writeToFile:**(NSString *)*fileName* **ofType:**(NSString *)*type*

Writes document data of type *docType* to the file at path *fileName*, returning whether the operation was successful. This method invokes **dataRepresentationOfType:** and is indirectly invoked whenever the document file is saved. It uses NSData's **writeToFile:atomically:** method to write to the file.

This method is one of the location-based primitives. Subclasses can override this method instead of overriding **dataRepresentationOfType:** to write document data to the file system as an NSData object after creating that object from internal data structures. Subclasses that handle file packages such as RTFD or that treat locations of files as anything other than paths should override this method. Override

implementations of this method should ensure that they filter document data appropriately using NSPasteboard's filtering services.

See also: – **loadDataRepresentation:ofType:**, – **readFromFile:ofType:**

NSDocumentController

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: AppKit/NSDocumentController.h

Class Description

An NSDocumentController object manages an application's documents. As the first-responder target of New and Open menu commands, it creates and opens documents and tracks them throughout a session of the application.When opening documents, an NSDocumentController runs and manages the modal Open panel. It saves and closes documents when applications are terminated and it responds to NSWorkspace methods when documents are opened or printed from the workspace. NSDocumentControllers also maintain and manage the mappings of document types, extensions, and NSDocument subclasses as specified in the NSTypes property loaded from the custom info property list (**CustomInfo.plist**). One instance of NSDocumentController is shared among all objects of an application.

You can use various NSDocumentController methods to get a list of the current documents, get the current document (which is the document whose window is currently key), get documents based on a given file name or window, and to find out about a document's extension, type, display name, and document class.

A document-based application can use the default NSDocumentController instance provided by the Application Kit. This instance not only creates and manages documents, reacts appropriately to Workspace and application events, much as a typical application delegate would. If you require additional behavior that's unrelated to documents, such as loading About panels and managing user preferences, you should have a simple custom controller object handle these chores rather than a subclass of NSDocumentController.

When an application is quitting, the NSDocumentController is asked before the application delegate (if one exists) to approve the closing.

For more on the architecture of document-based applications, and the procedure for implementing, see the concepts "“Document-Based Application Architecture”" and "“Implementing a Document-Based Application”", currently in the NSDocument class description.

Method Types

Obtaining the shared instance

+ sharedDocumentController

Creating and opening documents

– makeDocumentWithContentsOfFile:ofType:
– makeUntitledDocumentOfType:
– openDocumentWithContentsOfFile:display:
– openUntitledDocumentOfType:display:
– setShouldCreateUI:
– shouldCreateUI

Managing the Open panel

– runModalOpenPanel:forTypes:
– fileNamesFromRunningOpenPanel
– currentDirectory

Closing documents

– closeAllDocuments
– reviewUnsavedDocumentsWithAlertTitle:cancellable:

Responding to action messages

– newDocument:
– openDocument:
– saveAllDocuments:

Managing documents

– documents
– currentDocument
– documentClassForType:
– documentForFileName:
– documentForWindow:
– hasEditedDocuments

Managing document types

– displayNameForType:
– fileExtensionsFromType:
– typeFromFileExtension:

Validating menu items

– validateMenuItem:

Class Methods

sharedDocumentController

+ (id)**sharedDocumentController**

Returns the shared NSDocumentController instance. If one doesn't exist yet, it is created. Initialization reads in the document types from the NSTypes property list (in **CustomInfo.plist**), registers the instance

for NSWorkspaceWillPowerOffNotifications, and turns on the flag indicating that document user interfaces should be visible. You should always obtain your application's NSDocumentController using this method.

See also: – **setShouldCreateUI:**

Instance Methods

closeAllDocuments

– (BOOL)**closeAllDocuments**

Attempts to close all documents owned by the receiver after asking users if they want to save edited documents. Returns whether all documents were closed. This method is invoked in **reviewUnsavedDocumentsWithAlertTitle:cancellable:** when users choose the Review Unsaved option.

See also: – **canCloseDocument** (NSDocument)

currentDirectory

– (NSString *)**currentDirectory**

Returns the directory path to be used as the starting point in the Open panel, whenever one of the following is a valid directory (in this order):

- The directory location where the current document was last saved
- The last directory visited in the Open panel
- The user's home directory

See also: – **documentForFileName:**

currentDocument

– (id)**currentDocument**

Returns the NSDocument object associated with the main window.

See also: – **documentForFileName:**, – **documentForWindow:**, – **documents**

displayNameForType:

– (NSString *)**displayNameForType:**(NSString *)*docType*

Returns the descriptive name for the document type (*docType*), which is often part of the document's window title. This returned value is associated with the NSTypeName key in the NSType property list.If there is no such value, *docType* is returned.

See also: – **fileExtensionsFromType:**, – **typeFromFileExtension:**

documentClassForType:

– (Class)**documentClassForType:**(NSString *)*docType*

Returns the NSDocument subclass associated with document type *docType*, as defined in the NSType property of the custom info property list. The document type must be one the document can read. If the class cannot be found, returns **nil**.

See also: – **displayNameForType:**, – **fileExtensionsFromType:**, – **typeFromFileExtension:**

documentForFileName:

– (id)**documentForFileName:**(NSString *)*fileName*

Returns the NSDocument object for the file in which the document data is stored. The *fileName* argument is a fully qualified path in the file system. Returns **nil** if no document can be found.

See also: – **documentForWindow:**, – **documents**

documentForWindow:

– (id)**documentForWindow:**(NSWindow *)*window*

Returns the NSDocument object whose window controller (NSWindowController) owns the window object *window*; returns **nil** if *window* is itself **nil**, if *window* has no window controller, or if the window controller does not have an association with an NSDocument.

See also: – **currentDocument**, – **documentForFileName:**, – **documents**

documents

– (NSArray *)**documents**

Returns the NSDocument objects managed by the receiver. If there are currently no documents, it returns an empty NSArray.

See also: – **currentDocument**, – **documentForFileName:**, – **documentForWindow:**

fileExtensionsFromType:

– (NSArray *)**fileExtensionsFromType:**(NSString *)*docType*

Returns the allowable file extensions (as NSString objects) for document type *docType*. The first string in the returned NSArray is typically the most common extension.

See also: – **displayNameForType:**, – **typeFromFileExtension:**

fileNamesFromRunningOpenPanel

– (NSArray *)**fileNamesFromRunningOpenPanel**

Returns a selection of files chosen by the user in the Open panel. Each file in the returned NSArray is a fully qualified path to the file's location in the file system. This method is invoked by **openDocument:** and it invokes **runModalOpenPanel:forTypes:** after initializaing the Open panel (which includes getting the starting directory with **currentDirectory**). Returns **nil** if the user cancels the Open panel or makes no selection.

hasEditedDocuments

– (BOOL)**hasEditedDocuments**

Returns whether the receiver has any documents with unsaved changes.

See also: – **documents**

makeDocumentWithContentsOfFile:ofType:

– (id)**makeDocumentWithContentsOfFile:**(NSString *)*fileName* **ofType:**(NSString *)*docType*

Creates and returns an NSDocument object for document type *docType* from the contents of the file at *fileName*, which must be a fully qualified path. The returned object is not retained. Returns **nil** if the NSDocument subclass for *docType* couldn't be determined or if the object couldn't be created. This method

invokes NSDocument's **initWithContentsOfFile:ofType:** and is invoked by **openDocumentWithContentsOfFile:display:**.

See also: – **makeUntitledDocumentOfType:**, – **openDocument:**

makeUntitledDocumentOfType:

– (id)**makeUntitledDocumentOfType:**(NSString *)*type*

Creates and returns an NSDocument object for document type *docType*. The returned object is not retained. Returns **nil** if the NSDocument subclass for *docType* couldn't be determined or if the object couldn't be created. This method invokes NSDocument's **init** and is invoked by **openUntitledDocumentOfType: display:**.

See also: – **makeDocumentWithContentsOfFile:ofType:**, – **newDocument:**

newDocument:

– (void)**newDocument:**(id)*sender*

An action method invoked the New menu command, it creates a new NSDocument object and adds it to the list of such objects managed by the receiver. It invokes **openUntitledDocumentOfType:display:** with the document type (first argument) being the first one specified in the NSType property (defined in **CustomInfo.plis**t); the document type determines the NSDocument sublcass used to instantiate the document object.

See also: – **openDocument:**

openDocument:

– (void)**openDocument:**(id)*sender*

An action method invoked the Open menu command, it runs the modal Open panel and, based on the selected filenames, creates one or more NSDocument object from the contents of the files; it adds these objects to the list of NSDocument objects managed by the receiver. This method invokes **openDocumentWithContentsOfFile:display:**, which actually creates the NSDocument objects.

See also: – **fileNamesFromRunningOpenPanel**, – **newDocument:**

openDocumentWithContentsOfFile:display:

– (id)**openDocumentWithContentsOfFile:**(NSString *)*fileName* **display:**(BOOL)*flag*

Returns an NSDocument object created from the contents of the file at *fileName* (an absolute path) and displays it if *flag* is YES. The returned object is not retained, but is added to the receiver's list of managed documents. Returns **nil** if the object could not be created, typically because *fileName* does not point to a valid file or because there is no NSDocument sublcass for the document type (as indicated by the file extension). Even if *flag* is YES, the document is not displayed if **shouldCreateUI** returns NO. This method invokes **makeDocumentWithContentsOfFile:ofType:** to obtain the created NSDocument object.

See also: – **openDocument:**, – **openUntitledDocumentOfType:display:**, – **setShouldCreateUI:**

openUntitledDocumentOfType:display:

– (id)**openUntitledDocumentOfType:**(NSString *)*docType* **display:**(BOOL)*display*

Returns an NSDocument object instantiated from the NSDocument subclass required by document type *docType* and displays it if *flag* is YES. The returned object is not retained, but is added to the receiver's list of managed documents. Returns **nil** if the object could not be created, typically because no NSDocument sublcass could be found for *docType*. Even if *flag* is YES, the document is not displayed if **shouldCreateUI** returns NO. This method invokes **makeUntitledDocumentOfType:** to obtain the created NSDocument object.

See also: – **newDocument:**, – **openDocumentWithContentsOfFile:display:**, – **setShouldCreateUI:**

reviewUnsavedDocumentsWithAlertTitle:cancellable:

– (BOOL)**reviewUnsavedDocumentsWithAlertTitle:**(NSString *)*title* **cancellable:**(BOOL)*flag*

Displays an attention panel (a dialog) asking users if they want to review unsaved documents, quit regardless of unsaved documents, or (if *flag* is YES) if they want to cancel the impending save-and-terminate operation. Returns YES if the application is to quit and NO if otherwise (used only when the application is terminating). If the user selects the Review Unsaved option, **closeAllDocuments** is invoked. This method is invoked when users choose the Quit menu command and when the computer power is being turned off (in which case, *flag* is NO).

runModalOpenPanel:forTypes:

– (int)**runModalOpenPanel:**(NSOpenPanel *)*openPanel*
forTypes:(NSArray *)*extensions*

Invokes NSOpenPanel's **runModalForTypes:** passing the file *extensions* associated with a document type. This method is invoked by the **fileNamesFromRunningOpenPanel** method. Subclasses can override this method if they want specialized Open panel behavior.

saveAllDocuments:

– (void)**saveAllDocuments:**(id)*sender*

As the action method invoked by the Save All command, saves all open documents of the application that need to be saved.

See also: – **saveDocument:** (NSDocument)

setShouldCreateUI:

– (void)**setShouldCreateUI:**(BOOL)*flag*

Sets whether the window controllers (NSWindowControllers) of a document should be created when the document is created. When a window controller is created, it loads the nib file containing the window it manages. Often *flag* is set to NO for scripting or searching operations involving the document's data.

See also: – **shouldCreateUI**

shouldCreateUI

– (BOOL)**shouldCreateUI**

Returns whether the window controllers (NSWindowControllers) of a document should be created when the document is created.

See also: – **setShouldCreateUI:**

typeFromFileExtension:

– (NSString *)**typeFromFileExtension:**(NSString *)*fileExtension*

Returns the document type associated with files having extension *fileExtension*.

See also: – **displayNameForType:**, – **fileExtensionsFromType:**

validateMenuItem:

– (BOOL)**validateMenuItem:**(NSMenuItem *)*anItem*

Validates menu item *anItem*, returning YES if it should be enabled, NO otherwise. As implemented, if *anItem* is the Save All menu item, returns YES if there are any edited documents. Subclasses can override this method to perform additional validations.

NSDPSContext

Inherits From: NSGraphicsContext : NSObject

Conforms To: NSObject (NSObject)

Declared In: AppKit/NSDPSContext.h

Class Description

The NSDPSContext class is the programmatic interface to objects that represent Display PostScript System *contexts*. A context can be thought of as a *destination* to which PostScript code is sent for execution. Each Display PostScript context contains its own complete PostScript environment including its own local VM (PostScript Virtual Memory). Every context has its own set of stacks, including an operand stack, graphics state stack, dictionary stack, and execution stack. Every context also contains a **FontDirectory** which is local to that context, plus a **SharedFontDirectory** that is shared across all contexts. There are three built-in dictionaries in the dictionary stack. From top to bottom, they are **userdict**, **globaldict**, and **systemdict**. **userdict** is private to the context, while **globaldict** and **systemdict** are shared by all contexts. **globaldict** is a modifiable dictionary containing information common to all contexts. **systemdict** is a read-only dictionary containing all the PostScript operators.

At any time there is the notion of the *current context*. The current context for the current thread may be set using **setCurrentContext:.**

NSDPSContext objects by default write their output to a specified *data* destination. This is used for printing, faxing, and for generation of saved EPS (Encapsulated PostScript) code. The means to create contexts that interact with displays are platform-specific.

The NSApplication object creates an NSDPSContext by default.

NSDPSContext Objects and Display PostScript System Context Records

When an NSDPSContext object is created, it creates and manages a *DPSContext* record. Programmers familiar with the client side C function interface to the Display PostScript System can access the DPSContext record by sending a **context** message to an NSDPSContext object. You can then operate on this context record using any of the functions or single operator functions defined in the Display PostScript System client library. Conversely, you can create an NSDPSContext object from a DPSContext record with the **DPSContextObject** function, as defined in “Client Library Functions”. You can then work with the created NSDPSContext object using any of the methods described here.

General Exception Conditions

A variety of exceptions can be raised from NSDPSContext. In most cases, exceptions are raised because of errors returned from the Display PostScript Server. Exceptions are listed under “Types and Constants.” Also see the *Display PostScript System, Client Library Reference Manual*, by Adobe Systems Incorporated, for more details on Display PostScript System error names and their possible causes.

Method Types

Initializing a context

– initWithMutableData:forDebugging:languageEncoding:nameEncoding:textProc:errorProc:

Testing the drawing destination

– isDrawingToScreen

Accessing context data

– mutableData

Setting and identifying the current context

+ currentContext
+ setCurrentContext:
– DPSContext

Controlling the context

– flush
– interruptExecution
– notifyObjectWhenFinishedExecuting:
– resetCommunication
– wait

Managing returned text and errors

+ stringForDPSError:
– errorProc
– setErrorProc:
– setTextProc:
– textProc

Sending raw data

– printFormat:
– printFormat:arguments:
– writeData:
– writePostScriptWithLanguageEncodingConversion:

Managing binary object sequences

– awaitReturnValues
– writeBOSArray:count:ofType:
– writeBOSNumString:length:ofType:scale:
– writeBOSString:length:
– writeBinaryObjectSequence:length:
– updateNameMap

Managing chained contexts

– chainChildContext:
– childContext
– parentContext
– unchainContext

Controlling the wait cursor

– startWaitCursorTimer
– setWaitCursorEnabled:
– isWaitCursorEnabled

Debugging aids

+ areAllContextsOutputTraced
+ areAllContextsSynchronized
+ setAllContextsOutputTraced:
+ setAllContextsSynchronized:
– isOutputTraced
– isSynchronized
– setOutputTraced:
– setSynchronized:

Class Methods

areAllContextsOutputTraced

+ (BOOL)**areAllContextsOutputTraced**

Returns YES if the data flowing between the application's contexts and their destinations is copied to diagnostic output.

areAllContextsSynchronized

+ (BOOL)**areAllContextsSynchronized**

Returns YES if all NSPDSContext objects invoke the wait method after sending each batch of output.

currentContext

+ (id)**currentContext**

Returns the **id** for the current context of the current thread.

setAllContextsOutputTraced:

+ (void)**setAllContextsOutputTraced:**(BOOL)*flag*

Causes the data (PostScript code, return values, and so forth) flowing between the all the application's contexts and their destinations to be copied to diagnostic output.

setAllContextsSynchronized:

+ (void)**setAllContextsSynchronized:**(BOOL)*flag*

Causes the **wait** method to be invoked each time an NSDPSContext object sends a batch of output to its destination.

setCurrentContext:

+ (void)**setCurrentContext:**(NSGraphicsContext *)*context*

Installs *context* as the current context of the current thread.

stringForDPSError:

+ (NSString *)**stringForDPSError:**(const DPSBinObjSeqRec *)*error*

Returns a string representation of *error.*

Instance Methods

DPSContext

– (DPSContext)**DPSContext**

Returns the corresponding DPScontext.

awaitReturnValues

– (void)**awaitReturnValues**

Waits for all return values from the result table.

chainChildContext:

– (void)**chainChildContext:**(NSDPSContext *)*child*

Links *child* (and all of it's children) to the receiver as its chained context, a context that receives a copy of all PostScript code sent to the receiver.

childContext

– (NSDPSContext *)**childContext**

Returns the receiver's child context, or **nil** if none exists.

errorProc

– (DPSErrorProc)**errorProc**

Returns the context's error callback function.

flush

– (void)**flush**

Forces any buffered data to be sent to its destination.

initWithMutableData:forDebugging:languageEncoding:nameEncoding:textProc:errorProc:

– **initWithMutableData:**(NSMutableData *)*data*
forDebugging:(BOOL)*debug*
languageEncoding(DPSProgramEncoding)*langEnc*
nameEncoding(DPSNameEncoding)*nameEnc*
textProc:(DPSTextProc)*tProc*
errorProc:(DPSErrorProc)*errorProc*

Initializes a newly allocated NSDPSContext that writes its output to *data* using the language and name encodings specified by *langEnc* and *nameEnc*. The callback functions *tProc* and *errorProc* handle text and

errors generated by the context. If *debug* is YES, the output is given in human-readable form in which large structures (such as images) may be represented by comments.

interruptExecution

– (void)**interruptExecution**

Interrupts execution in the receiver's context.

isDrawingToScreen

– (BOOL)**isDrawingToScreen**

Returns YES if the drawing destination is the screen.

isOutputTraced

– (BOOL)**isOutputTraced**

Returns YES if the data flowing between the application's single context and its destination is copied to diagnostic output.

isSynchronized

– (BOOL)**isSynchronized**

Returns whether the **wait** method is invoked each time the receiver sends a batch of output to the server.

isWaitCursorEnabled

– (BOOL)**isWaitCursorEnabled**

Returns whether the wait cursor is enabled.

See also: **PScurrentwaitcursorenabled** (function)

mutableData

– (NSMutableData *)**mutableData**

Returns the receiver's data object.

notifyObjectWhenFinishedExecuting:

– (void)**notifyObjectWhenFinishedExecuting:**(id <NSDPSContextNotification>)*object*

Registers *object* to receive a **contextFinishedExecuting**: message when the NSDPSContext's destination is ready to receive more input. The registered *object* supports the NSDPSContextNotification protocol.

parentContext

– (NSDPSContext *)**parentContext**

Returns the receiver's parent context, or **nil** if none exists.

printFormat:

– (void)**printFormat:**(NSString *)*format,...*

Constructs a string from *format* and following string objects (in the manner of **printf**) and sends it to the context's destination.

printFormat:arguments:

– (void)**printFormat:**(NSString *)*format*
arguments:(va_list)*argList*

Constructs a string from *format* and *argList* (in the manner of **vprintf**) and sends it to the context's destination.

resetCommunication

– (void)**resetCommunication**

Discards any data that hasn't already been sent to its destination.

setErrorProc:

– (void)**setErrorProc:**(DPSErrorProc)*proc*

Sets the context's error callback function to *proc*.

setOutputTraced:

– (void)**setOutputTraced:**(BOOL)*flag*

Causes the data (PostScript code, return values, and so on) flowing between the application's single context and the Display PostScript server to be copied to diagnostic output.

setSynchronized:

– (void)**setSynchronized:**(BOOL)*flag*

Sets whether the **wait** method is invoked each time the receiver sends a batch of output to its destination.

setTextProc:

– (void)**setTextProc:**(DPSTextProc)*proc*

Sets the context's text callback function to *proc*.

setWaitCursorEnabled:

– (void)**setWaitCursorEnabled:**(BOOL)*flag*

Sets whether the wait cursor is enabled or disabled according to *flag*.

See also: **PSsetwaitcursorenabled** (function)

startWaitCursorTimer

– (void)**startWaitCursorTimer**

Generates a pseudo-event to start wait cursor timer.

See also: **– setWaitCursorEnabled:**

textProc

– (DPSTextProc)**textProc**

Returns the context's text callback function.

unchainContext

– (void)**unchainContext**

Unlinks the child context (and all of it's children) from the receiver's list of chained contexts.

updateNameMap

– (void)**updateNameMap**

Updates the context's name map from the client library's name map.

wait

– (void)**wait**

Waits until the NSDPSContext's destination is ready to receive more input.

writeBOSArray:count:ofType:

– (void)**writeBOSArray:**(const void *)*data*
count:(unsigned int)*items*
ofType:(DPSDefinedType)*type*

Write an array to the context's destination as part of a binary object sequence. The array is taken from *data* and consists of *items* items of type *type*.

writeBOSNumString:length:ofType:scale:

– (void)**writeBOSNumString:**(const void *)*data*
length:(unsigned int)*count*
ofType(DPSDefinedType)*type*
scale:(int)*scale*

Write a number string to the context's destination as part of a binary object sequence. The string is taken from *data* as described by *count*, *type*, and *scale.*

writeBOSString:length:

– (void)**writeBOSString:**(const void *)*data* **length:**(unsigned int)*bytes*

Write a string to the context's destination as part of a binary object sequence. The string is taken from *bytes* (a count) of *data.*

writeBinaryObjectSequence:length:

– (void)**writeBinaryObjectSequence:**(const void *)*data*
length:(unsigned int)*bytes*

Write a binary object sequence to the context's destination. The sequence consists of *bytes* (a count) of *data*.

writeData:

– (void)**writeData:**(NSData *)*buf*

Sends the PostScript data in *buf* to the context's destination.

writePostScriptWithLanguageEncodingConversion:

– (void)**writePostScriptWithLanguageEncodingConversion:**(NSData *)*buf*

Writes the PostScript data in *buf* to the context's destination. The data, formatted as plain text, encoded tokens, or a binary object sequence, is converted as necessary depending on the language encoding of the receiving context.

NSEPSImageRep

Inherits From: NSImageRep : NSObject

Conforms To: NSCoding (from NSImageRep)
NSCopying (from NSImageRep)
NSObject (from NSObject)

Declared In: AppKit/NSEPSImageRep.h

Class Description

An NSEPSImageRep is an object that can render an image from encapsulated PostScript code (EPS).

Like most other kinds of NSImageReps, an NSEPSImageRep is generally used indirectly, through an NSImage object. An NSImage must be able to choose between various representations of a given image. It also needs to provide an off-screen cache of the appropriate depth for any image it uses. It determines this information by querying its NSImageReps.

Thus to work with an NSImage, an NSEPSImageRep must be able to provide some information about its image. The size of the object is set from the bounding box specified in the EPS header comments. Use these methods, inherited from the NSImageRep class, to set the other attributes of the NSEPSImageRep:

setColorSpaceName:
setAlpha:
setPixelsHigh:
setPixelsWide:
setBitsPerSample:

Note that if these attributes aren't set, and an NSEPSImageRep is used in an NSImage with other representations, NSImage won't be able to select between them. In actual practice, this usually isn't a problem.

Method Types

Creating an NSEPSImageRep

+ imageRepWithData:
– initWithData:

Getting image data

– boundingBox
– EPSRepresentation

Drawing the image

– prepareGState

Class Methods

imageRepWithData:

+ (id)**imageRepWithData:**(NSData *)*epsData*

Creates a new NSEPSImageRep instance and then invokes **initWithData:** to initialize it with the contents of *epsData*. If the new object can't be initialized for any reason (for example, *epsData* doesn't contain EPS code), this method frees the receiver and returns **nil**. Otherwise, it returns a new instance of NSEPSImageRep.

The size of the object is set from the bounding box specified in the EPS header comments.

Instance Methods

EPSRepresentation

– (NSData *)**EPSRepresentation**

Returns the EPS representation of the image.

boundingBox

– (NSRect)**boundingBox**

Returns the rectangle that bounds the image. The rectangle is obtained from the "%%BoundingBox:" comment in the EPS header when the NSEPSImageRep is initialized.

See also: **+ imageRepWithData:**, **– initWithData:**

initWithData:

– (id)**initWithData:**(NSData *)*epsData*

Initializes the receiver, a newly allocated NSEPSImageRep object, with the contents of *epsData*. If the new object can't be initialized for any reason (for example, *epsData* doesn't contain EPS code), this method frees the receiver and returns **nil**. Otherwise, it returns **self**.

The size of the object is set from the bounding box specified in the EPS header comments.

prepareGState

– (void)**prepareGState**

Implemented by subclasses to initialize the graphics state before the image is drawn. NSEPSImageRep's **draw** method sends a **prepareGState** message just before rendering the EPS code. The default implementation of **prepareGState** does nothing.

NSEvent

Inherits From:	NSObject
Conforms To:	NSCoding NSCopying NSObject (NSObject)
Declared In:	AppKit/NSEvent.h

Class Description

An NSEvent object, or simply an *event*, contains information about an input action such as a mouse click or a key down. The Application Kit associates each such user action with a window, reporting the event to the application that created the window. The NSEvent object contains pertinent information about each event, such as where the mouse was located or which character was typed. As the application receives events, it temporarily places them in a buffer called the *event queue*. When the application is ready to process an event, it takes one from the queue.

NSEvents are typically passed up the application's *responder chain*, a series of objects that stand in line for event messages and untargeted action messages, as described in the NSResponder class specification. When the NSApplication object retrieves an event from the event queue, it dispatches the event to the appropriate NSWindow by invoking **sendEvent:**. The NSWindow then passes the event to its first responder in an event message such as **mouseDown:** or **keyDown:**, and the event gets passed on up the responder chain until some object handles it. In the case of a mouse-down event, a **mouseDown:** message is sent to the NSView where the user clicked the mouse; if it doesn't handle the event itself, the NSView relays the message to its next responder.

Most events follow this same path: from the windowing system to the application's event queue, and from there to the appropriate objects in the application. Though it rarely need do so, an application can also create an event from scratch and insert it into the event queue for distribution, or send it directly to its destination in an event message. The newly created events can be added to the event queue by invoking NSWindow's (or NSApplication's) **postEvent:atStart:** method.

While most events are distributed automatically through the responder chain, sometimes an object needs to retrieve events explicitly—for example, while handling mouse-dragged events. NSWindow and NSApplication define the method **nextEventMatchingMask:untilDate:inMode:dequeue:**, which allows an object to retrieve events of specific types. The nature of the retrieved event can then be ascertained by invoking NSEvent instance methods—**type**, **window**, and so on. All types of events are associated with an NSWindow; the **window** method returns this object. The location of a mouse event within the window's coordinate system is given by **locationInWindow**, and the time of the event by **timestamp**. The

modifierFlags method returns an indication of which modifier keys (Command, Control, Shift, and so on) the user held down while the event occurred.

The **type** method returns an NSEventType value that identifies the sort of event. The different types of events fall into five groups:

- Keyboard events
- Mouse events
- Tracking-rectangle and cursor-update events
- Periodic events
- Other events

Some of these groups comprise several NSEventType constants, others only one. The following sections discuss the groups, along with the corresponding NSEventType constants.

Keyboard Events

Among the most common events sent to an application are direct reports of the user's keyboard actions, identified by these NSEventType constants:

- NSKeyDown. The user generated a character or characters by pressing a key.
- NSKeyUp. The key was released.
- NSFlagsChanged. The user pressed or released a modifier key, or turned Alpha Lock on or off.

Of these, key-down events are the most useful to an application. When a **type** message returns NSKeyDown, the next step is typically to get the characters generated by the key-down using the **characters** method.

Key-up events are used less frequently since they follow almost automatically when there's been a key-down event. And because NSEvent's **modifierFlags** method returns the state of the modifier keys regardless of the type of event, applications normally don't need to receive flags-changed events; they're useful only for applications that have to keep track of the state of these keys at all times.

For more information on keyboard events, see "Key Events" under the Class Description in the NSResponder class specification and "Input Management" in the NSTextView class specification.

Mouse Events

Mouse events are generated by changes in the state of the mouse buttons and by changes in the position of the mouse cursor on the screen. This category consists of:

- NSLeftMouseDown, NSLeftMouseUp, NSRightMouseDown, NSRightMouseUp. "Mouse-down" means the user pressed the button; "mouse-up" means the user released it. If the mouse has just one button, only left mouse events are generated. By sending a **clickCount** message to the event, you can determine whether the mouse event was a single click, double click, and so on.

- NSLeftMouseDragged, NSRightMouseDragged. The user moved the mouse with one or more buttons down. NSLeftMouseDragged events are generated when the mouse is moved with its left mouse button down or with both buttons down, and NSRightMouseDragged when it's moved with just the right button down. A mouse with a single button generates only left mouse-dragged events. A series of mouse-dragged events is always preceded by a mouse-down event and followed by a mouse-up event.
- NSMouseMoved. The user moved the mouse without holding down either mouse button. Mouse-moved events are normally not tracked, as they quickly flood the event queue; use NSWindow's **setAcceptsMouseMovedEvents:** to turn on tracking of mouse movements.

Mouse-dragged and mouse-moved events are generated repeatedly as long as the user keeps moving the mouse. If the mouse is stationary, neither type of event is generated until the mouse moves again.

Note: Neither the OpenStep specification nor the Rhapsody implementation specifies facilities for the third button of a three-button mouse.

See "Mouse Events" under "Event Handling" in the NSView class specification for more information on mouse events.

Tracking-Rectangle and Cursor-Update Events

Because following the mouse's movements precisely is an expensive operation, the Application Kit provides a less intensive mechanism for tracking the location of the mouse. It does this by allowing the application to define regions of the screen, called *tracking rectangles*, that generate events when the cursor enters or leaves them. The event types are NSMouseEntered and NSMouseExited, and they're generated when the application has asked the Window Server to set a tracking rectangle in a window, typically by using NSView's **addTrackingRect:owner:userData:assumeInside:** method. A window can have any number of tracking rectangles; NSEvent's **trackingNumber** method identifies the rectangle that triggered the event.

A special kind of tracking event is the NSCursorUpdate event. This type is used to implement NSView's cursor-rectangle mechanism. An NSCursorUpdate event is generated when the cursor has crossed the boundary of a predefined rectangular area. Applications rarely use NSCursorUpdate events directly, instead using NSView's far more convenient methods.

See "Tracking Rectangles and Cursor Rectangles" under "Event Handling" in the NSView class specification for more information.

Periodic Events

An event of type NSPeriodic simply notifies an application that a certain time interval has elapsed. By using the NSEvent class method **startPeriodicEventsAfterDelay:withPeriod:**, an application can register to receive periodic events and have them placed in its event queue at a certain frequency. When the application no longer needs them, the flow of periodic events can be turned off by invoking **stopPeriodicEvents**. An application can have only one stream of periodic events active for each thread. Unlike keyboard and mouse

events, periodic events aren't dispatched to an NSWindow. The application must retrieve them explicitly using **nextEventMatchingMask:untilDate:inMode:dequeue:**, typically in a modal loop.

Periodic events are particularly useful in situations where input events aren't generated. For example, when the user holds the mouse down over a scroll button but doesn't move it, no events are generated after the mouse-down event. The scrolling mechanism then has to start and use a stream of periodic events to keep the document scrolling at a reasonable pace until the user releases the mouse. When a mouse-up event occurs, the scrolling mechanism terminates the periodic event stream.

Other Events

The remaining event types—NSAppKitDefined, NSSystemDefined, and NSApplicationDefined—are less structured, containing only generic subtype and data fields. These three types are extensions to the OpenStep specification, so you shouldn't use them in portable code (periodic events are also implemented in this manner, but are in the specification). Of the three miscellaneous event types, only NSApplicationDefined is of real use to application programs. It allows the application to generate totally custom events and insert them into the event queue. Each such event can have a subtype and two additional codes to differentiate it from others. **otherEventWithType:...** creates one of these events, and the **subtype**, **data1**, and **data2** methods return the information specific to these events.

Adopted Protocols

NSCoding

– encodeWithCoder:
– initWithCoder:

NSCopying

– copyWithZone:
– copy

Method Types

Creating events

+ keyEventWithType:location:modifierFlags:timestamp:windowNumber:context:characters:charactersIgnoringModifier:isARepeat:keyCode:
+ mouseEventWithType:location:modifierFlags:timestamp:windowNumber:context:eventNumber:clickCount:pressure:
+ enterExitEventWithType:location:modifierFlags:timestamp:windowNumber:context:eventNumber:trackingNumber:userData:
+ otherEventWithType:location:modifierFlags:timestamp:windowNumber:context:subtype:data1:data2:

Requesting and stopping periodic events

+ startPeriodicEventsAfterDelay:withPeriod:
+ stopPeriodicEvents

Getting general event information

– context
– locationInWindow
– modifierFlags
– timestamp
– type
– window
– windowNumber

Getting key event information

– characters
– charactersIgnoringModifiers
– isARepeat
– keyCode

Getting mouse event information

– clickCount
– eventNumber
– pressure

Getting tracking-rectangle event information

– eventNumber
– trackingNumber
– userData

Getting custom event information

– data1
– data2
– subtype

Class Methods

enterExitEventWithType:location:modifierFlags:timestamp: windowNumber:context:eventNumber:trackingNumber:userData:

+ (NSEvent *)**enterExitEventWithType:**(NSEventType)*type*
location:(NSPoint)*location*
modifierFlags:(unsigned int)*flags*
timestamp:(NSTimeInterval)*time*
windowNumber:(int)*windowNumber*
context:(NSDPSContext *)*context*
eventNumber:(int)*eventNumber*
trackingNumber:(int)*trackingNumber*
userData:(void *)*userData*

Returns a new NSEvent object describing a tracking-rectangle or cursor-update event. *type* must be one of the following, else an NSInvalidArgumentException is raised:

NSMouseEntered
NSMouseExited
NSCursorUpdate

location, *flags*, *time*, *windowNumber*, and *context* are as described under **keyEventWithType:...**. Arguments specific to mouse tracking events are:

eventNumber is an identifier for the new event. It's normally taken from a counter for mouse events, which continually increases as the application runs.

trackingNumber is the number that identifies the tracking rectangle. This identifier is the same returned by NSView's **addTrackingRect:owner:userData:assumeInside:**.

userData is data arbitrarily associated with the tracking rectangle when it was set up using NSView's **addTrackingRect:owner:userData:assumeInside:**.

See also: – **eventNumber**, – **trackingNumber**, – **userData**

keyEventWithType:location:modifierFlags:timestamp:windowNumber: context:characters:charactersIgnoringModifier:isARepeat:keyCode:

+ (NSEvent *)**keyEventWithType:**(NSEventType)*type*
  location:(NSPoint)*location*
  modifierFlags:(unsigned int)*flags*
  timestamp:(NSTimeInterval)*time*
  windowNumber:(int)*windowNum*
  context:(NSDPSContext *)*context*
  characters:(NSString *)*characters*
  charactersIgnoringModifiers:(NSString *)*unmodCharacters*
  isARepeat:(BOOL)*repeatKey*
  keyCode:(unsigned short int)*code*

Returns a new NSEvent object describing a key event. *type* must be one of the following, else an NSInvalidArgumentException is raised:

NSKeyDown
NSKeyUp
NSFlagsChanged

location is the mouse location in the base coordinate system of the window specified by *windowNumber*.

flags is an integer bit field containing any of these modifier key masks, combined using the C bitwise OR operator:

NSAlphaShiftKeyMask
NSShiftKeyMask
NSControlKeyMask
NSAlternateKeyMask
NSCommandKeyMask
NSNumericPadKeyMask
NSHelpKeyMask
NSFunctionKeyMask

time is the time the event occurred in seconds since system startup. How to get this value varies with the platform.

windowNumber identifies the PostScript window device associated with the event, which is associated with the NSWindow that will receive the event.

context is the Display PostScript context of the event.

characters is a string of characters associated with the key event. Though most key events contain only one character, it is possible for a single keypress to generate a series of characters.

unmodCharacters is the string of characters generated by the key event as if no modifier key had been pressed (except for Shift). This is useful for getting the "basic" key value in a hardware-independent manner.

repeatKey is YES if the key event is a repeat caused by the user holding the key down, NO if the key event is new.

code identifies the keyboard key associated with the key event. Its value is hardware-dependent.

See also: – **characters**, – **charactersIgnoringModifiers**, – **isARepeat**, – **keyCode**

mouseEventWithType:location:modifierFlags:timestamp: windowNumber:context:eventNumber:clickCount:pressure:

+ (NSEvent *)**mouseEventWithType:**(NSEventType)*type*
  location:(NSPoint)*location*
  modifierFlags:(unsigned int)*flags*
  timestamp:(NSTimeInterval)*time*
  windowNumber:(int)*windowNum*
  context:(NSDPSContext *)*context*
  eventNumber:(int)*eventNumber*
  clickCount:(int)*clickNumber*
  pressure:(float)*pressure*

Returns a new NSEvent object describing a mouse-down, -up, -moved, or -dragged event. *type* must be one of the following, else an NSInvalidArgumentException is raised:

NSLeftMouseDown
NSLeftMouseUp
NSRightMouseDown
NSRightMouseUp
NSMouseMoved
NSLeftMouseDragged
NSRightMouseDragged

location, *flags*, *time*, *windowNumber*, and *context* are as described under **keyEventWithType:...**.

eventNumber is an identifier for the new event. It's normally taken from a counter for mouse events, which continually increases as the application runs.

clickNumber is the number of mouse clicks associated with the mouse event.

pressure is a value from 0.0 to 1.0 indicating the pressure applied to the input device on a mouse event, used for an appropriate device such as a graphics tablet. For devices that aren't pressure-sensitive, the value

should be either 0.0 or 1.0. How to determine whether the input device is pressure-sensitive depends on the platform.

See also: – **clickCount**, – **eventNumber**, – **pressure**

otherEventWithType:location:modifierFlags:timestamp: windowNumber:context:subtype:data1:data2:

+ (NSEvent *)**otherEventWithType:**(NSEventType)*type*
location:(NSPoint)*location*
modifierFlags:(unsigned int)*flags*
timestamp:(NSTimeInterval)*time*
windowNumber:(int)*windowNum*
context:(NSDPSContext *)*context*
subtype:(short int)*subtype*
data1:(int)*data1*
data2:(int)*data2*

Returns a new NSEvent object describing a custom event. *type* must be one of the values below, else an NSInvalidArgumentException is raised. Your code should only create events of type NSApplicationDefined.

NSAppKitDefined (Apple extension to the OpenStep specification)
NSSystemDefined (Apple extension to the OpenStep specification)
NSApplicationDefined (Apple extension to the OpenStep specification)
NSPeriodic

location, *flags*, *time*, *windowNumber*, and *context* are as described under **keyEventWithType:...**. Arguments specific to mouse tracking events are:

subtype further differentiates custom events of type NSAppKitDefined, NSSystemDefined, and NSApplicationDefined. NSPeriodic events don't use this attribute.

data1 and *data2* contain additional data associated with the event. NSPeriodic events don't use these attributes.

See also: – **subtype**, – **data1**, – **data2**

startPeriodicEventsAfterDelay:withPeriod:

+ (void)**startPeriodicEventsAfterDelay:**(NSTimeInterval)*delaySeconds* **withPeriod:**(NSTimeInterval)*periodSeconds*

Begins generating periodic events for the current thread every *periodSeconds*, after a delay of *delaySeconds*. Raises an NSInternalInconsistencyException if periodic events are already being generated for the current thread. This method is typically used in a modal loop while tracking mouse-dragged events.

See also: + **stopPeriodicEvents**

stopPeriodicEvents

+ (void)**stopPeriodicEvents**

Stops generating periodic events for the current thread and discards any periodic events remaining in the queue. This message is ignored if periodic events aren't currently being generated.

See also: + **startPeriodicEventsAfterDelay:withPeriod:**

Instance Methods

characters

– (NSString *)**characters**

Returns the characters associated with the receiving key-up or key-down event. These characters are derived from a keyboard mapping that associates various key combinations with Unicode characters. Raises an NSInternalInconsistencyException if sent to any other kind of event.

See also: – **charactersIgnoringModifiers**, + **keyEventWithType:location:modifierFlags:timestamp: windowNumber:context:characters:charactersIgnoringModifier:isARepeat:keyCode:**

charactersIgnoringModifiers

– (NSString *)**charactersIgnoringModifiers**

Returns the characters generated by the receiving key event as if no modifier key (except for Shift) applies. Raises an NSInternalInconsistencyException if sent to a non-key event. The return value of this method is meaningless for an NSFlagsChanged event.

This method is useful for determining "basic" key values in a hardware-independent manner, enabling such features as keyboard equivalents and mnemonics defined in terms of modifier keys plus character keys. For example, to determine if the user typed Alt-s, you don't have to know whether Alt-s generates a German

double ess, an integral sign, or a section symbol. You simply examine the string returned by this method along with the event's modifier flags, checking for "s" and NSAlternateKeyMask.

See also: – **characters**, – **modifierFlags**, + **keyEventWithType:location:modifierFlags:timestamp: windowNumber:context:characters:charactersIgnoringModifier:isARepeat:keyCode:**

clickCount

– (int)**clickCount**

Returns the number of mouse clicks associated with the receiver, a mouse-down or -up event. Raises an NSInternalInconsistencyException if sent to a non-mouse event.

The return value of this method is meaningless for events other than mouse-down or -up events.

See also: + **mouseEventWithType:location:modifierFlags:timestamp:windowNumber:context: eventNumber:clickCount:pressure:**

context

– (NSDPSContext *)**context**

Returns the Display PostScript context of the receiving event.

data1

– (int)**data1**

Returns additional data associated with the receiving event. Raises an NSInternalInconsistencyException if sent to an event not of type NSAppKitDefined, NSSystemDefined, NSApplicationDefined, or NSPeriodic.

NSPeriodic events don't use this attribute.

See also: – **data2**, – **subtype**, + **otherEventWithType:location:modifierFlags:timestamp: windowNumber:context:subtype:data1:data2**:

data2

– (int)**data2**

Returns additional data associated with the receiving event. Raises an NSInternalInconsistencyException if sent to an event not of type NSAppKitDefined, NSSystemDefined, NSApplicationDefined, or NSPeriodic.

NSPeriodic events don't use this attribute.

See also: – **data1**, – **subtype**, + **otherEventWithType:location:modifierFlags:timestamp: windowNumber:context:subtype:data1:data2:**

eventNumber

– (int)**eventNumber**

Returns the counter value of the latest mouse or tracking-rectangle event; every system-generated mouse and tracking-rectangle event increments this counter. Raises an NSInternalInconsistencyException if sent to any other type of event.

See also: + **enterExitEventWithType:location:modifierFlags:timestamp:windowNumber:context: eventNumber:trackingNumber:userData:**,
+ **mouseEventWithType:location:modifierFlags:timestamp:windowNumber:context: eventNumber:clickCount:pressure:**

isARepeat

– (BOOL)**isARepeat**

Returns YES if the receiving key event is a repeat caused by the user holding the key down, NO if the key event is new. Raises an NSInternalInconsistencyException if sent to a non-key event.

The return value of this method is meaningless for NSFlagsChanged events.

See also: + **keyEventWithType:location:modifierFlags:timestamp:windowNumber:context: characters:charactersIgnoringModifier:isARepeat:keyCode:**

keyCode

– (unsigned short int)**keyCode**

Returns the code for the keyboard key associated with the receiving key event. Its value is hardware-dependent. Raises an NSInternalInconsistencyException if sent to a non-key event.

See also: + **keyEventWithType:location:modifierFlags:timestamp:windowNumber:context: characters:charactersIgnoringModifier:isARepeat:keyCode:**

locationInWindow

– (NSPoint)**locationInWindow**

Returns the receiving event's location in the base coordinate system of the associated window.

See also: – **window**

modifierFlags

– (unsigned int)**modifierFlags**

Returns an integer bit field indicating the modifier keys in effect for the receiving event. You can examine individual flag settings using the C bitwise AND operator with these predefined masks:

NSAlphaShiftKeyMask
NSShiftKeyMask
NSControlKeyMask
NSAlternateKeyMask
NSCommandKeyMask
NSNumericPadKeyMask
NSHelpKeyMask
NSFunctionKeyMask

pressure

– (float)**pressure**

Returns a value between 0.0 and 1.0 indicating the pressure applied to the input device (used for appropriate devices). For devices that aren't pressure-sensitive, the value is either 0.0 or 1.0. How to determine whether the input device is pressure-sensitive depends on the platform. Raises an NSInternalInconsistencyException if sent to a non-mouse event.

See also: + **mouseEventWithType:location:modifierFlags:timestamp:windowNumber:context: eventNumber:clickCount:pressure:**

subtype

– (short int)**subtype**

Returns the subtype of the receiving custom event. Raises an NSInternalInconsistencyException if sent to an event not of type NSAppKitDefined, NSSystemDefined, NSApplicationDefined, or NSPeriodic.

NSPeriodic events don't use this attribute.

See also: – **data1**, – **data2**, + **otherEventWithType:location:modifierFlags:timestamp:windowNumber:context:subtype:data1:data2:**

timestamp

– (NSTimeInterval)**timestamp**

Returns the time the event occurred in seconds since system startup.

trackingNumber

– (int)**trackingNumber**

Returns the identifier of the tracking rectangle for a tracking-rectangle event. Raises an NSInternalInconsistencyException if sent to any other type of event.

See also: + **enterExitEventWithType:location:modifierFlags:timestamp:windowNumber:context:eventNumber:trackingNumber:userData:**

type

– (NSEventType)**type**

Returns the type of the receiving event, one of:

NSLeftMouseDown	NSKeyDown
NSLeftMouseUp	NSKeyUp
NSRightMouseDown	NSFlagsChanged
NSRightMouseUp	NSAppKitDefined (Apple extension to the OpenStep specification)
NSMouseMoved	NSSystemDefined (Apple extension to the OpenStep specification)
NSLeftMouseDragged	NSApplicationDefined (Apple extension to the OpenStep specification)

NSRightMouseDragged	NSPeriodic
NSMouseEntered	NSCursorUpdate
NSMouseExited	

userData

– (void *)**userData**

Returns data associated with a tracking-rectangle event, assigned to the tracking rectangle when it was set up using NSView's **addTrackingRect:owner:userData:assumeInside:**. Raises an NSInternalInconsistencyException if sent to any other type of event.

See also: + **enterExitEventWithType:location:modifierFlags:timestamp:windowNumber:context: eventNumber:trackingNumber:userData:**

window

– (NSWindow *)**window**

Returns the window object associated with the event. A periodic event, however, has no window; in this case the return value is undefined.

See also: – **windowNumber**

windowNumber

– (int)**windowNumber**

Returns the identifier for the PostScript window device associated with the event. A periodic event, however, has no window; in this case the return value is undefined.

See also: – **window**

NSFileWrapper

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: AppKit/NSFileWrapper.h

Class Description

An NSFileWrapper holds a file's contents in dynamic memory. In this role it enables a document object to embed a file, treating it as a unit of data that can be displayed as an image (and possibly edited in place), saved to disk, or transmitted to another application. It can also store an icon for representing the file in a document or in a dragging operation.

Instances of this class are referred to as *file wrapper objects*, and when no confusion will result, merely as *file wrappers*. A file wrapper can be one of three specific types: a *regular file wrapper*, which holds the contents of a single actual file; a *directory wrapper*, which holds a directory and all of the files or directories within it; or a *link wrapper*, which simply represents a symbolic link in the file system (sometimes called a shortcut or alias). Some NSFileWrapper methods apply only to a specific type, and raise an exception if sent to a file wrapper of the wrong type. To determine the type of a file wrapper, use the **isRegularFile**, **isDirectory**, and **isSymbolicLink** methods.

You can create a file wrapper from data in memory using **initWithSerializedRepresentation:** or from data on disk using **initWithPath:**. Both create the appropriate type of file wrapper based on the nature of the serialized representation or of the file on disk. Three convenience methods each create a file wrapper of a specific type: **initRegularFileWithContents:**, **initDirectoryWithFileWrappers:**, and **initSymbolicLinkWithDestination:**. Because each initialization method creates file wrappers of different types or states, they're all designated initializers for this class—subclasses must meaningfully override them all as necessary.

Memory and Disk Representations

Because the purpose of a file wrapper is to represent files in memory, it's very loosely coupled to any disk representation. A file wrapper doesn't record the path to the disk representation of its contents. This allows you to save the same file wrapper with different paths, but it also requires you to record those paths if you want to update the file wrapper from disk later. NSFileWrapper allows you to set a preferred filename for save operations and records the last filename it was actually saved to; the **preferredFilename** and **filename** methods return these names. This feature is more important for directory wrappers, though, and so is discussed under "Working with Directory Wrappers" below.

A file wrapper stores file system information (such as modification time and access permissions), which it updates when reading from disk and uses when writing files to disk. The **fileAttributes** method returns this information in the format described in the NSFileManager class specification. You can also set the file attributes using the **setFileAttributes:** method.

When saving a file wrapper to disk, you typically determine the directory you want to save it in, then append the preferred filename to that directory path and use **writeToFile:atomically:updateFilenames:**, which saves the file wrapper's contents and updates the file attributes. You can save a file wrapper under a different name if you wish, but this may result in the recorded filename differing from the preferred filename, depending on how you invoke the **writeToFile:...** method.

Besides saving its contents to disk, a file wrapper can re-read them from disk when necessary. The **needsToBeUpdatedFromPath:** method determines whether a disk representation may have changed, based on the file attributes stored the last time the file was read or written. If the file wrapper's modification time or access permissions are different from those of the file on disk, this method returns YES. You can then use **updateFromPath:** to re-read the file from disk.

Finally, to transmit a file wrapper to another process or system (for example, over a distributed objects connection or through the pasteboard), you use the **serializedRepresentation** method to get an NSData object containing the file wrapper's contents in the NSFileContentsPboardType format. You can safely transmit this representation over whatever channel you desire. The recipient of the representation can then reconstitute the file wrapper using the **initWithSerializedRepresentation:** method.

Working with Directory Wrappers

A directory wrapper contains other file wrappers (of any type), and allows you to access them by keys derived from their preferred filenames. You can add any type of file wrapper to a directory wrapper with **addFileWrapper:** or **addFileWithPath:**, and remove it with **removeFileWrapper:**. The convenience methods **addRegularFileWithContents:preferredFilename:** and **addSymbolicLinkWithDestination: preferredFilename:** allow you to add regular file and link wrappers while also setting their preferred names.

A directory wrapper stores its contents in an NSDictionary, which you can retrieve using the **fileWrappers** method. The keys of this dictionary are based on the preferred filenames of each file wrapper contained in the directory wrapper. There exist, then, three identifiers for a file wrapper within a directory wrapper:

- Preferred filename. This doesn't uniquely identify the file wrapper, but the following identifiers are always based on it.
- Dictionary key. This is always equal to the preferred name when there are no other file wrappers of the same preferred name in the same directory wrapper. Otherwise, it's a string made by adding a unique prefix to the preferred filename (note that the same file wrapper can have a different dictionary key for each directory wrapper that contains it). You use the dictionary key to retrieve the file wrapper object in memory, in order to get its contents or its filename (to update it from disk). You can get a file wrapper's dictionary key by sending a **keyForFileWrapper:** message to the directory wrapper that contains it.

- Filename. This is usually based on the preferred filename, but isn't necessarily the same as it or the dictionary key. You use the filename to update a single file wrapper relative to the path of the directory wrapper that contains it. Note that the filename may change whenever you save a directory wrapper containing the file wrapper (particularly if the file wrapper has been added to several different directory wrappers); thus, you should always retrieve the filename from the file wrapper itself each time you need it rather than caching it.

When working with the contents of a directory wrapper, you can use a dictionary enumerator to retrieve each file wrapper and perform whatever operation you need. Note that with the exceptions of saving and updating, a directory file wrapper defines no recursive operations for its contents. To set the file attributes for all contained file wrappers, or to perform any other such operation, you must define a recursive method that examines the type of each file wrapper and invokes itself anew for any directory wrapper it encounters.

Method Types

Initializing a file wrapper

– initWithPath:
– initDirectoryWithFileWrappers:
– initRegularFileWithContents:
– initSymbolicLinkWithDestination:
– initWithSerializedRepresentation:

Writing to a file or serializing

– writeToFile:atomically:updateFilenames:
– serializedRepresentation

Checking a file wrapper's type

– isRegularFile
– isDirectory
– isSymbolicLink

Setting attributes

– setFilename:
– filename
– setPreferredFilename:
– preferredFilename
– setIcon:
– icon
– setFileAttributes:
– fileAttributes

Updating

– needsToBeUpdatedFromPath:
– updateFromPath:

Modifying a directory wrapper

– addFileWrapper:
– removeFileWrapper:
– addFileWithPath:
– addRegularFileWithContents:preferredFilename:
– addSymbolicLinkWithDestination:preferredFilename:
– fileWrappers
– keyForFileWrapper:

Inspecting a regular file wrapper

– regularFileContents

Inspecting a link wrapper

– symbolicLinkDestination

Instance Methods

addFileWithPath:

– (NSString *)**addFileWithPath:**(NSString *)*path*

Adds a new file wrapper to the receiving directory wrapper. Initializes the new file wrapper with **initWithPath:** using *path* as the argument, then adds the new file wrapper by invoking **addFileWrapper:** . Returns the dictionary key used for the newly added file wrapper within the directory wrapper. Raises an NSInternalInconsistencyException if sent to a regular file or link wrapper.

See also: – **addRegularFileWithContents:preferredFilename:**, – **addSymbolicLinkWithDestination: preferredFilename:**, – **removeFileWrapper:**, – **fileWrappers**

addFileWrapper:

– (NSString *)**addFileWrapper:**(NSFileWrapper *)*wrapper*

Adds *wrapper* to the receiving directory wrapper. Returns the dictionary key used for *wrapper* within the directory wrapper. Raises an NSInternalInconsistencyException if sent to a regular file or link wrapper, or an NSInvalidArgumentException if *wrapper* doesn't have a preferred name (set using **setPreferredName:**).

See also: – **addFileWithPath:**, – **addRegularFileWithContents:preferredFilename:**, – **addSymbolicLinkWithDestination:preferredFilename:**, – **removeFileWrapper:**, – **fileWrappers**

addRegularFileWithContents:preferredFilename:

– (NSString *)**addRegularFileWithContents:**(NSData *)*contents* **preferredFilename:** (NSString *)*filename*

Adds a new regular file wrapper to the receiving directory wrapper. Initializes the new file wrapper with **initRegularFileWithContents:** using *contents* as the argument, sets its preferred name with **setPreferredName:** using *filename* as the argument, then adds the new file wrapper by invoking **addFileWrapper:**. Returns the dictionary key used for the newly added file wrapper within the directory wrapper. Raises an NSInternalInconsistencyException if sent to a regular file or link wrapper, or an NSInvalidArgumentException if *filename* is **nil** or empty.

See also: – **addFileWithPath:**, – **addSymbolicLinkWithDestination:preferredFilename:**, – **removeFileWrapper:**, – **fileWrappers**

addSymbolicLinkWithDestination:preferredFilename:

– (NSString *)**addSymbolicLinkWithDestination:**(NSString *)*path* **preferredFilename:** (NSString *)*filename*

Adds a new link wrapper to the receiving directory wrapper. Initializes the new link wrapper with **initSymbolicLinkWithDestination:** using *path* as the argument, sets its preferred name with **setPreferredName:** using *filename* as the argument, then adds the new link wrapper by invoking **addFileWrapper:**. Returns the dictionary key used for the newly added link wrapper within the directory wrapper. Raises an NSInternalInconsistencyException if sent to a regular file or link wrapper, or an NSInvalidArgumentException if *filename* is **nil** or empty.

See also: – **addFileWithPath:**, – **addFileWrapper:**, – **addRegularFileWithContents: preferredFilename:**, – **removeFileWrapper:**, – **fileWrappers**

fileAttributes

– (NSDictionary *)**fileAttributes**

Returns the file attributes last read from disk or set using **setFileAttributes:**. These attributes are used whenever the file wrapper is saved using **writeToFile:atomically:updateFilenames:**. See the NSFileManager class specification for information on the contents of the attributes dictionary.

filename

– (NSString *)**filename**

Returns the filename for the last known disk representation of the receiver, or **nil** if the receiver has no filename. The filename is used for record-keeping purposes only, and is set automatically when the file wrapper is created from disk using **initWithPath:** and when it’s saved to a disk using **writeToFile:**

atomically:updateFilenames: (although this method allows you to request that the filename not be updated).

See also: – **preferredFilename**, – **setFilename:**

fileWrappers

– (NSDictionary *)**fileWrappers**

Returns the file wrappers contained in a directory wrapper. Raises an NSInternalInconsistencyException if sent to a regular file or link wrapper. See “Working with Directory Wrappers” in the class description for information on the dictionary.

See also: – **filename**, – **addFileWrapper:**

icon

– (NSImage *)**icon**

Returns an image that can be used to represent the file wrapper to the user, or **nil** if the file wrapper has none. You don’t have to use this image; for example, a file viewer typically looks up icons automatically based on file extensions, and so wouldn’t need this image. Similarly, if a file wrapper represents an image file, you can display the image directly rather than a file icon.

See also: – **setIcon:**

initDirectoryWithFileWrappers:

– (id)**initDirectoryWithFileWrappers:**(NSDictionary *)*wrappers*

Initializes a newly allocated NSFileWrapper as a directory wrapper containing *wrappers*. The new directory wrapper has no filename or associated disk representation until you save it using **writeToFile:atomically: updateFilenames:**. It’s also initialized with open permissions; anyone can read, write, or change directory to the disk representations that it saves.

If any file wrapper in *wrappers* doesn’t have a preferred name, its preferred name is automatically set to its corresponding dictionary key in *wrappers*.

This method is a designated initializer for the NSFileWrapper class. Returns **self**.

See also: – **setPreferredFilename:**, – **filename**, – **setFileAttributes:**

initRegularFileWithContents:

– (id)**initRegularFileWithContents:**(NSData *)*contents*

Initializes a newly allocated NSFileWrapper as a regular file wrapper with *contents*. The new file wrapper has no filename or associated disk representation until you save it using **writeToFile:atomically: updateFilenames:**. It's also initialized with open permissions; anyone can read or write the disk representations that it saves.

This method is a designated initializer for the NSFileWrapper class. Returns **self**.

See also: – **setPreferredFilename:**, – **filename**, – **fileAttributes**

initSymbolicLinkWithDestination:

– (id)**initSymbolicLinkWithDestination:**(NSString *)*path*

Initializes a newly allocated NSFileWrapper as a link wrapper pointing to *path*. The new file wrapper has no filename or associated disk representation until you save it using **writeToFile:atomically: updateFilenames:**. It's also initialized with open permissions; anyone can read or write the disk representations that it saves.

This method is a designated initializer for the NSFileWrapper class. Returns **self**.

See also: – **setPreferredFilename:**, – **filename**, – **fileAttributes**

initWithPath:

– (id)**initWithPath:**(NSString *)*path*

Initializes a newly allocated NSFileWrapper with the file or directory at *path*, setting its type to regular file, directory, or link wrapper based on the type of that file and caching the file's attributes. Also sets the receiver's preferred filename and recorded filename to the last component of *path*. If *path* identifies a directory, this method recursively creates file wrappers for each file or directory within that directory.

This method is a designated initializer for the NSFileWrapper class. Returns **self**.

See also: – **setPreferredFilename:**, – **filename**, – **fileAttributes**

initWithSerializedRepresentation:

– (id)**initWithSerializedRepresentation:**(NSData *)*data*

Initializes a newly allocated NSFileWrapper with *data*, setting its type to regular file, directory, or link wrapper based on the nature of that data and reading the file attributes from the data as well. *data* is a serialized representation of a file's or directory's contents in the format used for the pasteboard type

NSFileContentsPboardType. Data of this format is returned by such methods as **serializedRepresentation** or NSAttributedString's **RTFDFromRange:**.

The new file wrapper has no filename or associated disk representation until you save it using **writeToFile: atomically:updateFilenames:**. This method is a designated initializer for the NSFileWrapper class. Returns **self**.

See also: – **setPreferredFilename:**, – **filename**, – **fileAttributes**

isDirectory

– (BOOL)**isDirectory**

Returns YES if the receiver is a directory wrapper, NO otherwise.

See also: – **isRegularFile**, – **isSymbolicLink**

isRegularFile

– (BOOL)**isRegularFile**

Returns YES if the receiver is a regular file wrapper, NO otherwise.

See also: – **isDirectory**, – **isSymbolicLink**

isSymbolicLink

– (BOOL)**isSymbolicLink**

Returns YES if the receiver is a link wrapper, NO otherwise.

See also: – **isDirectory**, – **isRegularFile**

keyForFileWrapper:

– (NSString *)**keyForFileWrapper:**(NSFileWrapper *)*wrapper*

Returns the key by which the receiving directory wrapper stores *wrapper* in its dictionary (as returned by the **fileWrappers** method). This is not necessarily the filename for *wrapper*. Raises an NSInternalInconsistencyException if sent to a regular file or link wrapper.

See also: – **filename**

needsToBeUpdatedFromPath:

– (BOOL)**needsToBeUpdatedFromPath:**(NSString *)*path*

Returns YES if the receiver's contents on disk may have changed, NO otherwise. For a regular file wrapper, this is determined by comparing the modification time and access permissions of the file or directory at *path* against those of the receiver. For a link wrapper, this is determined by checking whether the destination path has changed (not by checking the modification time or access attributes of the destination). For a directory, this is determined as needed recursively for each file wrapper contained in the directory; added or removed files also count as changes.

See also: – **updateFromPath:**, – **fileAttributes**

preferredFilename

– (NSString *)**preferredFilename**

Returns the file wrapper's preferred filename. This name is used as the default dictionary key and filename when a file wrapper is added to a directory wrapper. However, if another file wrapper with the same preferred name already exists in the directory wrapper when the receiver is added, the dictionary key and filename assigned may differ from the preferred filename.

See also: – **setPreferredFilename:**, – **filename**

regularFileContents

– (NSData *)**regularFileContents**

Returns the contents of the receiving regular file wrapper. Raises an NSInternalInconsistencyException if sent to a directory or link wrapper.

removeFileWrapper:

– (void)**removeFileWrapper:**(NSFileWrapper *)*wrapper*

Removes *wrapper* from the receiving directory wrapper and releases it. Raises an NSInternalInconsistencyException if sent to a regular file or link wrapper.

See also: – **addFileWithPath:**, – **addFileWrapper:**, – **addRegularFileWithContents: preferredFilename:**, – **addSymbolicLinkWithDestination:preferredFilename:**, – **fileWrappers**

serializedRepresentation

– (NSData *)**serializedRepresentation**

Returns the receiver's contents as an opaque collection of data, in the format used for the pasteboard type NSFileContentsPboardType.

See also: **– initWithSerializedRepresentation:**

setFileAttributes:

– (void)**setFileAttributes:**(NSDictionary *)*attributes*

Sets the file attributes that are applied whenever the file wrapper is saved using **writeToFile:atomically: updateFilenames:** to *attributes*. See the NSFileManager class specification for information on the contents of the attributes dictionary.

See also: **– fileAttributes**

setFilename:

– (void)**setFilename:**(NSString *)*filename*

Sets the filename for the disk representation of the receiver to *filename*. The filename is used for record-keeping purposes only, and is set automatically when the file wrapper is saved to a disk using **writeToFile:atomically:updateFilenames:** (although this method allows you to request that the filename not be updated). You should rarely need to invoke this method.

Raises an NSInvalidArgumentException if *filename* is **nil** or empty.

See also: **– setPreferredFilename:**, **– filename**

setIcon:

– (void)**setIcon:**(NSImage *)*anImage*

Sets the image that can be used to represent the file wrapper to the user to *anImage*. You don't have to use this image; for example, a file viewer typically looks up icons automatically based on file extensions, and so wouldn't need this image. Similarly, if a file wrapper represents an image file, you can display the image directly rather than a file icon.

See also: **– icon**

setPreferredFilename:

– (void)**setPreferredFilename:**(NSString *)*filename*

Sets the receiver's preferred filename to *filename*. This name is used as the default dictionary key and filename when a file wrapper is added to a directory wrapper. However, if another file wrapper with the same preferred name already exists in the directory wrapper when the receiver is added, the dictionary key and filename assigned may differ from the preferred filename. Raises an NSInvalidArgumentException if *filename* is **nil** or empty.

See also: – **preferredFilename**, – **addFileWrapper:**, – **setFilename:**

symbolicLinkDestination

– (NSString *)**symbolicLinkDestination**

Returns the actual path represented by the receiving link wrapper. Raises NSInternalInconsistencyException if sent to a regular file or directory wrapper.

updateFromPath:

– (BOOL)**updateFromPath:**(NSString *)*path*

Re-reads the file wrapper's information from the file or directory at *path*, including contents or link destination, icon, file attributes. For a directory wrapper, the contained file wrappers are also sent **updateFromPath:** messages. If files in the directory on disk have been added or removed, corresponding file wrappers are released or created as needed. Returns YES if updating actually occurred, NO if it wasn't necessary.

See also: – **needsToBeUpdatedFromPath:**, – **updateAttachmentsFromPath:** (NSAttributedString)

writeToFile:atomically:updateFilenames:

– (BOOL)**writeToFile:**(NSString *)*path*
atomically:(BOOL)*atomicFlag*
updateFilenames:(BOOL)*updateNamesFlag*

Writes the receiver's contents to a file or directory at *path*. Returns YES on success and NO on failure. If *atomicFlag* is YES, attempts to write the file safely so that an existing file at *path* is not overwritten, nor does a new file at *path* actually get created, unless the write is successful. If *updateNamesFlag* is YES and the contents are successfully written, changes the receiver's filename to the last component of *path*, and the filenames of any children of a directory wrapper to the filenames under which they're written to disk.

If you're executing a "save" or "save as" style operation, pass YES for *updateNamesFlag;* if you're executing a "save to" style operation, pass NO for *updateNamesFlag*.

See also: – **filename**

NSFont

Inherits From:	NSObject
Conforms To:	NSCoding NSCopying NSObject (NSObject)
Declared In:	AppKit/NSFont.h

Class Description

NSFont objects represent PostScript fonts to an application, providing access to characteristics of the font and assistance in laying out glyphs relative to one another. Font objects are also used to establish the current font when drawing in an NSView, using the **set** method.

You don't create Font objects using the **alloc** and **init** methods. Instead, you use one of the **fontWithName:** methods to look up an available font and alter its size or matrix to your needs. These methods check for an existing font object with the specified characteristics, returning it if there is one. Otherwise, they look up the font data requested and create the appropriate object. NSFont also defines a number of methods for getting standard system fonts, such as **systemFontOfSize:**, **userFontOfSize:**, and **messageFontOfSize:**.

Drawing Text with NSFonts

In most cases you draw text using an NSTextView object. You can also draw an NSString directly in an NSView using the methods **drawAtPoint:withAttributes:** and **drawInRect:withAttributes:**, which the Application Kit adds to NSString. These methods take an NSDictionary of attributes, as used by the NSAttributedString class, and apply them when drawing the string.

If you need to draw text using PostScript operators such as **show**, it's recommended that you set the current font using NSFont's **set** method, rather than the PostScript operators **setfont** or **selectfont**. This allows the Application Kit printing mechanism to record the fonts used in the PostScript output. If you absolutely must set the font using a PostScript operator, you can record the font with the Application Kit using the class method **useFont:**. See the description of that method for more information.

Getting Font Metrics

NSFont defines a number of methods for accessing a font's metrics information, when that information is available. Methods such as **boundingRectForGlyph:**, **boundingRectForFont**, **xHeight**, and so on, all correspond to standard font metrics information. See the various method descriptions for specific

information. You can also get a complete dictionary of font metrics using the **afmDictionary** method, or retrieve the original contents of the metrics file using **afmFileContents**.

Calculating Glyph Layout

The OPENSTEP extended text system handles many complex aspects of laying glyphs out. If you need to calculate layout for your own purposes, you can use several methods defined by NSFont. There are three basic kinds of glyph layout:

- Sequential, for running text
- Overstruck, for diacritics and other non-spacing marks
- Stacked, for certain non-Western scripts.

Sequential glyph layout

Sequential glyph layout is supported by the method **positionOfGlyph:precededByGlyph:isNominal:**. This method calculates the position of a glyph relative to glyph preceding it, using the glyph's width and kerning information if they're available. This is the most straightforward kind of glyph layout.

Overstruck glyph layout

Overstruck glyph layout is the most complex, as it requires detailed information about placement of many kinds of modifying marks. Generally, you have two characters:

- A *base glyph* which may be a character such as *a*
- A *non-spacing mark* which may be a diacritical mark such as an acute accent (*'*) or a cedilla *(¸)*.

OPENSTEP gives you a few methods for combining the two characters together, depending on whether the combination is a common one that the font has metrics for or whether the combination is an unusual one that you need to create on the fly. Try these methods in the following order, to get the best result:

- To see if the font has metrics placing the non-spacing mark directly over the base glyph, use the method **positionOfGlyph:struckOverGlyph:metricsExist:** and check the value returned in the *metricsExist* argument.

- To see if the font has metrics for placing the non-spacing mark over the base glyph's bounding rectangle, use the method **positionOfGlyph:struckOverRect:metricsExist:** and check the value returned in the *metricsExist* argument. (Use the method **boundingRectForGlyph:** to get the bounding rect for the base glyph.) Note that NSFont always sets *metricsExist* to NO and that this method is useful only if you're using a subclass of NSFont that overrides this method.

- To place the non-spacing mark over the base glyph in a legible but not necessarily pleasing manner, use the method **positionOfGlyph:forCharacter:struckOverRect:**. (Use the method **boundingRectForGlyph:** to get the bounding rect for the base glyph.) This method handles all the common non-spacing marks, such as an acute accent, tilde, or cedilla, for Latin script.

- To place a non-spacing mark over a base glyph of another font, also use the method **positionOfGlyph: forCharacter:struckOverRect:**. (Use the method **boundingRectForGlyph:** to get the bounding rect for the base glyph.)

If you need to place several non-spacing marks with respect to a base glyph, use the method **positionsForCompositeSequence:numberOfGlyphs:pointArray:**. This method accepts a C array containing the base glyph followed by all of its non-spacing marks, and calculates the positions for as many as of the marks as it can. To place the marks that this method can't handle, use the methods described above.

Stacked glyph layout

Stacked glyph layout is supported by the method **positionOfGlyph:withRelation:toBaseGlyph: totalAdvancement:metricsExist:**. Stacked glyphs often have special compressed forms, which standard font metrics don't account for. NSFont's implementation of this method simply abuts the bounding boxes of the two glyphs for approximate layout of the individual glyphs. Subclasses of NSFont can override this method to access any extra metrics information for more sophisticated layout of stacked glyphs.

Special Glyphs

NSFont defines two special glyphs. NSNullGlyph indicates no glyph at all, and is useful in some layout methods for calculating information that isn't relative to another glyph. For example, with **positionOfGlyph:precededByGlyph:isNominal:**, you can specify NSNullGlyph as the first glyph to get the nominal advancement of the preceding glyph.

The other special glyph is NSControlGlyph, which the text system maps onto control functions such as linefeed and tab. This glyph has no graphic representation and has no inherent advancement of its own. Instead, the text system examines the control character underlying the glyph to determine what kind of special layout it needs to perform.

Adopted Protocols

NSCoding

– encodeWithCoder:
– initWithCoder:

NSCopying

– copyWithZone:

Method Types

Creating arbitrary fonts

+ fontWithName:size:
+ fontWithName:matrix:

Creating user fonts

+ userFontOfSize:
+ userFixedPitchFontOfSize:

Creating system fonts

+ boldSystemFontOfSize:
+ controlContentFontOfSize:
+ menuFontOfSize:
+ messageFontOfSize:
+ paletteFontOfSize:
+ systemFontOfSize:
+ titleBarFontOfSize:
+ toolTipsFontOfSize:

Getting preferred fonts

+ setPreferredFontNames:
+ preferredFontNames

Using a font to draw

– set

Adding fonts to print operations

+ useFont:

Getting general font information

– encodingScheme
– isBaseFont
– isFixedPitch
– mostCompatibleStringEncoding

Getting information about glyphs

– glyphIsEncoded:
– glyphPacking
– glyphWithName:

Getting metrics information

– advancementForGlyph:
– afmDictionary
– afmFileContents
– ascender
– boundingRectForFont
– boundingRectForGlyph:
– capHeight
– descender
– italicAngle
– matrix
– maximumAdvancement
– pointSize
– underlinePosition
– underlineThickness
– widthOfString:
– widths
– xHeight

Getting font names

– displayName
– familyName
– fontName

Laying out sequential glyphs

– positionOfGlyph:precededByGlyph:isNominal:
– positionsForCompositeSequence:numberOfGlyphs:pointArray:

Laying out overstruck glyphs

– positionOfGlyph:forCharacter:struckOverRect:
– positionOfGlyph:struckOverGlyph:metricsExist:
– positionOfGlyph:struckOverRect:metricsExist:

Laying out stacked glyphs

– positionOfGlyph:withRelation:toBaseGlyph:totalAdvancement:metricsExist:

Setting user fonts

+ setUserFont:
+ setUserFixedPitchFont:

Getting corresponding device fonts

– printerFont
– screenFont

Class Methods

boldSystemFontOfSize:

+ (NSFont *)**boldSystemFontOfSize:**(float)*fontSize*

Returns the font used for standard interface items that are rendered in boldface type, in *fontSize*. This is available for backwards compatibility only and calls **titleBarFontOfSize:**. Use one of the specialized methods instead, such as **userFontOfSize:**, **userFixedPitchFontOfSize:**, **titleBarFontOfSize:**, **menuFontOfSize:**, **messageFontOfSize:**, **paletteFontOfSize:**, or **toolTipsFontOfSize:**.

See also: + **fontWithName:size:**

controlContentFontOfSize:

+ (NSFont *)**controlContentFontOfSize:**(float)*fontSize*

Returns the font used for the content of controls, in *fontSize*. For example, in a table, the user's input uses the control content font and the table's header uses another font.

See also: + **fontWithName:size:**

fontWithName:matrix:

+ (NSFont *)**fontWithName:**(NSString *)*typeface* **matrix:**(const float *)*fontMatrix*

Returns a font object for *typeface* and *fontMatrix*. A typeface is a fully specified family-face name, such as Helvetica-BoldOblique or Times-Roman (not a name as shown in the Font Panel). *fontMatrix* is a standard 6-element transformation matrix as used in the PostScript language, specifically with the **makefont** operator. In most cases, you can simply use **fontWithName:size:** to create standard scaled fonts.

You can use the defined value NSFontIdentityMatrix for [1 0 0 1 0 0]. Fonts created with a matrix other than NSFontIdentityMatrix *don't* automatically flip themselves in flipped views.

See also: – **isFlipped** (NSView)

fontWithName:size:

+ (NSFont *)**fontWithName:**(NSString *)*fontName* **size:**(float)*fontSize*

Returns a font object for *fontName* and *fontSize*. A typeface is a fully specified family-face name, such as Helvetica-BoldOblique or Times-Roman. *fontSize* is used to scale the font, and is equivalent to using a font matrix of [*fontSize* 0 0 *fontSize* 0 0] with **fontWithName:matrix:**. If you use a *fontSize* of 0.0, this method uses the default User Font size.

Fonts created with this method automatically flip themselves in flipped views. This method is the preferred means for creating fonts.

menuFontOfSize:

+ (NSFont *)**menuFontOfSize:**(float)*fontSize*

Returns the font used for menu items in *fontSize*.

See also: + **fontWithName:size:**

messageFontOfSize:

+ (NSFont *)**messageFontOfSize:**(float)*fontSize*

Returns the font used for standard interface items, such as button labels, menu items, and so on, in *fontSize*. This is equivalent to **systemFontOfSize:**.

See also: + **fontWithName:size:**

paletteFontOfSize:

+ (NSFont *)**paletteFontOfSize:**(float)*fontSize*

Returns the font used for palette window title bars.

See also: + **fontWithName:size:**, + **titleBarFontOfSize:**

preferredFontNames

+ (NSArray *)**preferredFontNames**

Returns the names of fonts that the Application Kit tries first when a character has no font specified, or when the font specified doesn't have a glyph for that character. If none of these fonts provides a glyph, the remaining fonts on the system are searched for a glyph.

See also: + **setPreferredFontNames:**

setPreferredFontNames:

+ (void)**setPreferredFontNames:**(NSArray *)*fontNames*

Sets the list of preferred font names to *fontNames*, and records them in the user defaults database for all applications. The Application Kit tries these fonts first when a character has no font specified, or when the

font specified doesn't have a glyph for that character. If none of these fonts provides a glyph, the remaining fonts on the system are searched for a glyph.

This method is useful for optimizing glyph rendering for uncommon scripts, by guaranteeing that appropriate fonts are searched first. For example, suppose you have three hundred Latin alphabet fonts and one Cyrillic alphabet font. When you read a document in Russian, you want it to find the Cyrillic font quickly. Ordinarily, the Application Kit will search for the Cyrillic font among all three hundred and one fonts. But if it is in the list of preferred fonts, the Cyrillic font will be one of the first searched.

See also: + **preferredFontNames**

setUserFixedPitchFont:

+ (void)**setUserFixedPitchFont:**(NSFont *)*aFont*

Sets the font used by default for documents and other text under the user's control, when that font should be fixed-pitch, to *aFont*, and records the font in the user defaults database for all applications.

See also: + **setUserFont:**, + **userFixedPitchFontOfSize:**

setUserFont:

+ (void)**setUserFont:**(NSFont *)*aFont*

Sets the font used by default for documents and other text under the user's control to *aFont*, and records the font in the user defaults database for all applications.

See also: + **setUserFixedPitchFont:**, + **userFontOfSize:**

systemFontOfSize:

+ (NSFont *)**systemFontOfSize:**(float)*fontSize*

Returns the font used for standard interface items, such as button labels, menu items, and so on, in *fontSize*. This is available for backwards compatibility only and calls **messageFontOfSize:**. Use one of the specialized methods instead, such as **userFontOfSize:**, **userFixedPitchFontOfSize:**, **titleBarFontOfSize:** , **menuFontOfSize:**, **messageFontOfSize:**, **paletteFontOfSize:**, or **toolTipsFontOfSize:**.

See also: + **boldSystemFontOfSize:**, + **userFontOfSize:**, + **userFixedPitchFontOfSize:**, + **fontWithName:size:**

titleBarFontOfSize:

+ (NSFont *)**titleBarFontOfSize:**(float)*fontSize*

Returns the font used for window title bars, in *fontSize*. This is equivalent to **boldSystemFontOfSize:**.

See also: + **paletteFontOfSize:**

toolTipsFontOfSize:

+ (NSFont *)**toolTipsFontOfSize:**(float)*fontSize*

Returns the font used for tool-tips labels, in *fontSize*.

See also: + **fontWithName:size:**

useFont:

+ (void)**useFont:**(NSString *)*fontName*

Records *fontName* as one used in the current print operation.

The NSFont class object keeps track of the fonts used in an NSView by recording each one that receives a **set** message. When the view is called upon to generate conforming PostScript language output (such as during printing), the NSFont class provides the list of fonts required for the **%%DocumentFonts** comment, as required by Adobe's Document Structuring Conventions.

useFont: augments this system by providing a way to register fonts that are included in the document but not set using NSFont's **set** method. For example, you might set a font by executing the **setfont** operator within a function created by the **pswrap** utility. In such a case, be sure to pair the use of the font with a **useFont:** message to register the font for listing in the document comments.

userFixedPitchFontOfSize:

+ (NSFont *)**userFixedPitchFontOfSize:**(float)*fontSize*

Returns the font used by default for documents and other text under the user's control (that is, text whose font the user can normally change), when that font should be fixed-pitch.

Note: The system does not guarantee that all the glyphs in a fixed-pitch font are the same width. For example certain Japanese fonts are dual-pitch, and other fonts may have non-spacing marks which can affect the display of other glyphs.

See also: + **userFontOfSize:**, + **fontWithName:size:**, + **setUserFixedPitchFont:**

userFontOfSize:

+ (NSFont *)**userFontOfSize:**(float)*fontSize*

Returns the font used by default for documents and other text under the user's control (that is, text whose font the user can normally change).

See also: + **userFixedPitchFontOfSize:**, + **fontWithName:size:**, + **setUserFont:**

Instance Methods

advancementForGlyph:

– (NSSize)**advancementForGlyph:**(NSGlyph)*aGlyph*

Returns the nominal spacing for *aGlyph*—the distance that the current point moves after showing the glyph—accounting for the receiving font's size. This spacing is given according to the glyph's movement direction, which is either strictly horizontal or strictly vertical.

See also: – **boundingRectForGlyph:**, – **maximumAdvancement**

afmDictionary

– (NSDictionary *)**afmDictionary**

Returns the receiving font's AFM information in dictionary form. It contains the following information under these keys, with all values as strings:

- NSAFMFamilyName
- NSAFMCapHeight
- NSAFMFontName
- NSAFMXHeight
- NSAFMFormatVersion
- NSAFMAscender
- NSAFMFullName
- NSAFMDescender
- NSAFMNotice
- NSAFMUnderlinePosition
- NSAFMVersion
- NSAFMUnderlineThickness
- NSAFMWeight
- NSAFMItalicAngle
- NSAFMEncodingScheme
- NSAFMMappingScheme
- NSAFMCharacterSet

For any other items, use the AFM file contents, as returned by **afmFileContents**.

afmFileContents

– (NSString *)**afmFileContents**

Returns the receiving font's AFM file as a string object.

ascender

– (float)**ascender**

Returns the top y coordinate of the receiving font's longest ascender.

See also: – **descender**, – **capHeight**, – **xHeight**

boundingRectForFont

– (NSRect)**boundingRectForFont**

Returns the receiving font's bounding rectangle, scaled to the font's size. The bounding rectangle is the union of the bounding rectangles of every glyph in the font.

See also: – **boundingRectForGlyph:**

boundingRectForGlyph:

– (NSRect)**boundingRectForGlyph:**(NSGlyph)*aGlyph*

Returns the bounding rectangle for *aGlyph*, scaled to the receiving font's size.

Note: Japanese fonts encoded with the scheme "EUC12-NJE-CFEncoding" do not have individual metrics or bounding boxes available for the glyphs above 127. For those glyphs, this method returns the bounding rectangle for the font, instead.

See also: – **boundingRectForFont**

capHeight

– (float)**capHeight**

Returns the receiving font's cap height.

See also: – **ascender**, – **descender**, – **xHeight**

descender

– (float)**descender**

Returns the bottom y coordinate of the receiving font's longest descender.

displayName

– (NSString *)**displayName**

Returns the name used to represent the receiving font in the user interface, typically localized for the user's language.

encodingScheme

– (NSString *)**encodingScheme**

Returns the name of the receiving font's encoding scheme, such as "AdobeStandardEncoding", "ISOLatin1Encoding", "FontSpecific", and so on.

familyName

– (NSString *)**familyName**

Returns the receiving font's family name; for example, "Times" or "Helvetica". This is the name that NSFontManager uses and may differ slightly from the AFM name.

See also: – **fontName**

fontName

– (NSString *)**fontName**

Returns the receiver's full font name, as used in PostScript language code; for example, "Times-Roman" or "Helvetica-Oblique".

See also: – **familyName**

glyphIsEncoded:

– (BOOL)**glyphIsEncoded:**(NSGlyph)*aGlyph*

Returns YES if the receiving font encodes *aGlyph*, NO if it doesn't contain it.

glyphPacking

– (NSMultibyteGlyphPacking) **glyphPacking**

Returns the best way to encode the font's glyphs into an array of bytes. The return value is one of the following:

- NSOneByteGlyphPacking
- NSJapaneseEUCGlyphPacking
- NSAsciiWithDoubleByteEUCGlyphPacking
- NSTwoByteGlyphPacking
- NSFourByteGlyphPacking

glyphWithName:

– (NSGlyph)**glyphWithName:**(NSString *)*glyphName*

Returns the encoded glyph named *glyphName*, or –1 if the receiving font contains no such glyph. Also returns –1 if the glyph named *glyphName* isn't encoded.

Note: Glyph names in fonts do not always accurately identify the glyph. If possible, look up the appropriate glyph on your own.

isBaseFont

– (BOOL)**isBaseFont**

Returns YES if the receiver is a PostScript base font, NO if it's a PostScript composite font composed of other base fonts.

isFixedPitch

– (BOOL)**isFixedPitch**

Returns YES if all glyphs in the receiving font have the same advancement, NO if any advancements differ.

Note: On the Mach platform, some Japanese fonts encoded with the scheme "EUC12-NJE-CFEncoding" return that they have the same advancement, but actually encode glyphs with one of two advancements. This is for historical compatibility. You may need to handle such fonts specially for some applications.

See also: – **advancementForGlyph:**

italicAngle

– (float)**italicAngle**

Returns the receiving font's italic angle, the amount that the font is slanted in degrees counterclockwise from the vertical, as read from its AFM file.

matrix

– (const float *)**matrix**

Returns the receiver's font matrix, a standard 6-element transformation matrix as used in the PostScript language, specifically with the **makefont** operator. In most cases, with a font of *fontSize*, this matrix is [*fontSize* 0 0 *fontSize* 0 0].

See also: + **fontWithName:matrix:**

maximumAdvancement

– (NSSize)**maximumAdvancement**

Returns the greatest advancement of any of the receiving font's glyphs. This is always either strictly horizontal or strictly vertical.

See also: – **advancementForGlyph:**

mostCompatibleStringEncoding

– (NSStringEncoding)**mostCompatibleStringEncoding**

Returns the string encoding that works best with the receiving font, where there are the fewest possible unmatched characters in the string encoding and glyphs in the font. You can use NSString's **dataUsingEncoding:** or **dataUsingEncoding:allowLossyConversion:** method to convert the string to this encoding.

If this method returns NSASCIIStringEncoding, it could not determine the correct encoding and assumed that the font can render only ASCII characters.

Note: This method works heuristically using well-known font encodings, so for nonstandard encodings it may not in fact return the optimal string encoding.

See also: – **widthOfString:**

pointSize

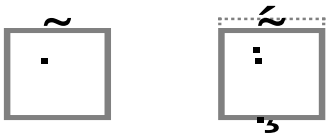
– (float)**pointSize**

Returns the receiving font's point size, or the effective vertical point size for a font with a nonstandard matrix.

positionOfGlyph:forCharacter:struckOverRect:

– (NSPoint)**positionOfGlyph:**(NSGlyph)*aGlyph*
forCharacter:(unichar)*aChar*
struckOverRect:(NSRect)*aRect*

Calculates and returns a suitable location for *aGlyph* to be drawn as a diacritic or non-spacing mark relative to *aRect*, assuming that *aGlyph* represents *aChar*. Returns NSZeroPoint if the location can't be calculated. The nature of *aChar* as one appearing above or below its base character determines the location returned. For example, in the first figure below, the gray tilde and box represent *aGlyph* and *aRect*, and the black dot is the point returned (defined relative to the origin of the *aRect*).



To place multiple glyphs with respect to a rectangle, work from the innermost glyphs to the outermost. As you calculate the position of each glyph, enlarge the rectangle to include the bounding rectangle of the glyph in preparation for the next glyph. The second figure shows a tilde, acute accent, and cedilla all placed in their appropriate positions with respect to a rectangle, with the acute accent placed relative to the expanded bounding box of the base rectangle and the tilde.

This method is the last fallback mechanism for performing minimally legible typography when metrics aren't available. Use it when **positionOfGlyph:struckOverGlyph:metricsExist:** indicates that metrics don't exist for the base glyph specified, or when you are combining glyphs from different fonts (for example, the base glyph is in a different font than the accent). It can account for the layout and placement of most Latin, Greek, and Cyrillic non-spacing marks. You should draw the glyph at the returned location, even if it's NSZeroRect.

positionOfGlyph:precededByGlyph:isNominal:

– (NSPoint)**positionOfGlyph:**(NSGlyph)*aGlyph*
precededByGlyph:(NSGlyph)*prevGlyph*
isNominal:(BOOL *)*flag*

Calculates and returns the location of *aGlyph* relative to *prevGlyph*, assuming that *prevGlyph* precedes it in the layout (*not* necessarily in the character stream). The point returned should be used relative to whatever location is used for *prevGlyph*. If *flag* is non-NULL, it's filled with NO if kerning tables are available and were used in the calculation; it is filled with YES if the default spacing is used.

Returns NSZeroPoint if either *aGlyph* or *prevGlyph* is NSControlGlyph or is invalid. Returns the nominal advancement of *prevGlyph* if *aGlyph* is NSNullGlyph.

This method is useful for sequential glyph placement when glyphs aren't drawn with a single PostScript operation.

positionOfGlyph:struckOverGlyph:metricsExist:

– (NSPoint)**positionOfGlyph:**(NSGlyph)*aGlyph*
struckOverGlyph:(NSGlyph)*baseGlyph*
metricsExist:(BOOL *)*flag*

Calculates and returns a suitable location for *aGlyph* to be drawn as a diacritic or non-spacing mark relative to *baseGlyph*. The point returned should be used relative to whatever location is used for *baseGlyph*. If *flag* is non-NULL it's filled with YES if font metrics are available, NO if they're not. If *flag* is returned as NO, the result isn't valid and shouldn't be used. In that case, use **positionOfGlyph:struckOverRect: metricsExist:** or **positionOfGlyph:forCharacter:struckOverRect:** to calculate a reasonable offset.

See also: – **positionsForCompositeSequence:numberOfGlyphs:pointArray:**, – **positionOfGlyph: struckOverRect:metricsExist:**

positionOfGlyph:struckOverRect:metricsExist:

– (NSPoint)**positionOfGlyph:**(NSGlyph)*aGlyph*
struckOverRect:(NSRect)*aRect*
metricsExist:(BOOL *)*flag*

Overridden by subclasses to calculate and return a suitable location for *aGlyph* to be drawn as a diacritic or non-spacing mark relative to *aRect*, provided metrics exist. Returns NSZeroRect if the location can't be determined. If *flag* is non-NULL it's filled with YES if font metrics are available, NO if they're not. If *flag* is returned as NO, the result isn't valid and shouldn't be used. In that case, use **positionOfGlyph: forCharacter:struckOverRect:** to calculate a reasonable offset.

Because current Postscript font metrics don't include support for generic placement relative to rectangles, NSFont's implementation of this method always returns NSZeroPoint and returns *flag* as NO.

positionOfGlyph:withRelation:toBaseGlyph:totalAdvancement:metricsExist:

– (NSPoint)**positionOfGlyph:**(NSGlyph)*aGlyph*
withRelation:(NSGlyphRelation)*relation*
toBaseGlyph:(NSGlyph)*baseGlyph*
totalAdvancement:(NSSizePointer)*offset*
metricsExist:(BOOL *)*flag*

Calculates and returns a suitable location for *aGlyph* to be drawn relative to *baseGlyph*, where *relation* is NSGlyphBelow or NSGlyphAbove. The point returned should be used relative to whatever location is used for *baseGlyph*. This method is useful for calculating the layout of stacked glyphs, found in some non-Western scripts.

If *offset* is non-NULL, this method sets it to the larger of the two glyphs' advancements, allowing for reasonable layout of following glyphs.

If *flag* is non-NULL, this method sets it to whether font metrics are available: YES if they are, NO if they're not. If metrics aren't available, the location is calculated as a simple stacking with no gap between *baseGlyph* and *aGlyph*. Current Postscript fonts do not contain appropriate font metrics, so this method always sets flag to NO. If you subclass NSFont to handle fonts that do contain metrics, override this method.

Note: *This method supports only horizontally laid-out base glyphs*.

positionsForCompositeSequence:numberOfGlyphs:pointArray:

– (int)**positionsForCompositeSequence:**(NSGlyph *)*glyphs*
numberOfGlyphs:(int)*numGlyphs*
pointArray:(NSPointArray)*points*

Calculates and fills *points* with the locations for *glyphs*, assuming that the first glyph is a base character and those following are non-spacing marks. These points should all be interpreted as relative to the location of the first glyph in *glyphs*. The storage block that *points* points to should be large enough for at least *numGlyphs* points. Returns the number of points that could be calculated.

If the number of points calculated is less than *numGlyphs*, the number of glyphs provided, you can use **positionOfGlyph:structOverRect:metricsExist:** to determine the positions for the remaining glyphs. When using that method, calculate the base rectangle for each glyph from the bounding rectangles and positions of all preceding glyphs.

printerFont

– (NSFont *)**printerFont**

When sent to a font object representing a scalable PostScript font, returns **self**. When sent to a font object representing a bitmapped screen font, returns its corresponding scalable PostScript font.

See also: – **screenFont**

screenFont

– (NSFont *)**screenFont**

When sent to a font object representing a scalable PostScript font, returns a bitmapped screen font matching the receiver in typeface and matrix (or size), or **nil** if such a font can't be found. When sent to a font object representing a bitmapped screen font, returns **nil**.

Note: Screen fonts are for direct use with the Window Server only. Never use them with Application Kit objects, such as in **setFont:** methods. Internally, the Application Kit automatically uses the corresponding screen font for a font object as long as the view is not rotated or scaled.

See also: – **printerFont**

set

– (void)**set**

Establishes the receiving font as the current font for PostScript **show** and other text-drawing operators. During a print operation, also records the font as used in the PostScript code emitted.

See also: + **useFont:**

underlinePosition

– (float)**underlinePosition**

Returns the baseline offset that should be used when drawing underlines with the receiving font, as determined by the font's AFM file. This value is usually negative, which must be considered when drawing in a flipped coordinate system.

See also: – **underlineThickness**

underlineThickness

– (float)**underlineThickness**

Returns the thickness that should be used when drawing underlines with the receiving font, as determined by the font's AFM file.

See also: – **underlinePosition**

widthOfString:

– (float)**widthOfString:**(NSString *)*aString*

Returns the *x* axis offset of the current point when *aString* is drawn with a PostScript **show** operator in the receiving font. This method performs lossy conversion of *aString* to the most compatible encoding for the receiving font.

Use this method only when you're sure all of *aString* can be rendered with the receiving font.

This method is for backwards compatibility only. In new code, use the Application Kit's string-drawing methods, as described under NSString Additions.

See also: – **mostCompatibleStringEncoding**

widths

– (float *)**widths**

Returns a C array of 256 **float**s, giving the unscaled width of each glyph in the font. This data is useful only for simple fonts without non-spacing marks, and doesn't account for Unicode-related issues at all.

This method is for backwards compatibility only. In new code, use **advancementForGlyph:** instead.

xHeight

– (float)**xHeight**

Returns the x-height of the receiving font.

See also: – **ascender**, – **descender**

NSFontManager

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: AppKit/NSFontManager.h

Class Description

NSFontManager is the center of activity for the font conversion system. It records the currently selected font, updates the Font Panel and Font menu to reflect the selected font, initiates font changes, and converts fonts in response to requests from text-bearing objects. In a more prosaic role, NSFontManager can be queried for the fonts available to the application, and for the particular attributes of a font, such as whether it's condensed or extended.

You normally set up a font manager and the Font Menu using Interface Builder. However, you can also do so programmatically by getting the shared font manager instance and having it create the standard Font menu at run time:

```
NSFontManager *fontManager = [NSFontManager sharedFontManager];
NSMenu *fontMenu = [fontManager fontMenu:YES];
```

You can then add the Font menu to your application's main menu. Once the Font menu is installed, your application automatically gains the functionality of both the Font menu and the Font Panel.

Recording the Selected Font

Any object that records fonts that the user can change should tell the font manager what the font of its selection is whenever it becomes the first responder and whenever its selection changes while it's the first responder. The object does so by sending the shared font manager a **setSelectedFont:isMultiple:** message. It should pass in the first font of the selection, along with a flag indicating whether there's more than one font.

The font manager uses this information to update the Font Panel and Font menu to reflect the selected font. For example, suppose the selected font is set as Helvetica Oblique 12.0 point. In this case the Font Panel selects that font and displays its name; the Font menu changes its Italic command to Unitalic; if there's no Bold variant of Helvetica available, the Bold menu item is disabled; and so on.

Initiating Font Changes

The user normally changes the font of the selection by manipulating the Font Panel and Font menu. These objects initiate the intended change by sending an action message to the font manager. There are four font-changing action methods:

– addFontTrait:
– removeFontTrait:
– modifyFont:
– modifyFontViaPanel:

The first three cause the font manager to query the sender of the message in order to determine which trait to add or remove, or how to modify the font. The last causes the font manager to use the settings in the Font Panel to modify the font. The font manager records this information and uses it in later requests to convert fonts, as described under “Responding to Font Changes”.

When the font manager receives an **addFontTrait:** or **removeFontTrait:** message, it queries the sender with a **tag** message, interpreting the return value as a trait mask for use with **convertFont:toHaveTrait:** or **convertFont:toNotHaveTrait:**, as described below under “Converting Fonts Manually”. The Italic and Bold Font menu commands, for example, have tags of NSItalicFontMask and NSBoldFontMask, respectively. See **convertFont:toHaveTrait:** for a list of trait mask values.

When the font manager receives a **modifyFont:** message, it queries the sender with a **tag** message and interprets the return value as a particular kind of conversion to perform, via the various conversion methods described under “Converting Fonts Manually”. For example, a button whose tag value is NSSizeUpFontAction causes the font manager’s **convertFont:** method to increase the size of the NSFont passed as the argument. See **modifyFont:** for a list of conversion tag values.

For **modifyFontViaPanel:**, the font manager sends the application’s Font Panel a **panelConvertFont:** message. The Font Panel in turn uses the font manager to convert the font provided according to the user’s choices. For example, if the user selects only the font family in the Font Panel (perhaps to Helvetica), then whatever fonts are provided to **panelConvertFont:**, only the family is changed: Courier Medium 10.0 point becomes Helvetica Medium 10.0 point, while Times Italic 12.0 point becomes Helvetica Oblique 12.0 point.

Responding to Font Changes

The font manager responds to a font-changing action method by sending a **changeFont:** action message up the responder chain. A text-bearing object that receives this message should have the font manager convert the fonts in its selection by invoking **convertFont:** for each font and using the NSFont object returned. **convertFont:** uses the information recorded by the font-changing action method, such as **addFontTrait:**, modifying the font provided appropriately. (There’s no way to explicitly set the font-changing action or trait; instead, you use the methods described under “Converting Fonts Manually”.)

This simple example assumes there’s only one font in the selection:

```
- (void)changeFont:(id)sender
{
    NSFont *oldFont = [self selectionFont];
    NSFont *newFont = [sender convertFont:oldFont];
    [self setSelectionFont:newFont];
    return;
}
```

Most text-bearing objects will have to scan the selection for ranges with different fonts, and invoke **convertFont:** for each one.

Font Trait Masks

NSFontManager categorizes fonts according to a small set of traits. You can convert fonts by adding and removing individual traits, and you can get a font with a specific combination of traits. The traits defined and available for your use are:

- NSBoldFontMask
- NSCompressedFontMask
- NSCondensedFontMask
- NSExpandedFontMask
- NSFixedPitchFontMask
- NSItalicFontMask
- NSNarrowFontMask
- NSNonStandardCharacterSetFontMask
- NSPosterFontMask
- NSSmallCapsFontMask
- NSUnboldFontMask
- NSUnitalicFontMask

These pairs of traits are mutually exclusive:

- NSCondensedFontMask and NSExpandedFontMask
- NSBoldFontMask and NSUnboldFontMask
- NSItalicFontMask and NSUnitalicFontMask

Converting Fonts Manually

NSFontManager defines a number of methods for explicitly converting particular traits and characteristics of a font. These methods are:

– convertFont:toFace:
– convertFont:toFamily:
– convertFont:toHaveTrait:
– convertFont:toNotHaveTrait:

– convertFont:toSize:
– convertWeight:ofFont:

Each returns a transformed version of the font provided, or the original font if it can't be converted. **convertFont:toFace:** and **convertFont:toFamily:** both alter the basic design of the font provided. The first method requires a fully-specified typeface name, such as "Times-Roman" or "Helvetica-BoldOblique", while the second expects only a family name, such as "Times" or "Helvetica".

convertFont:toHaveTrait: and **convertFont:toNotHaveTrait:** use trait masks to add or remove a single trait such as Italic, Bold, Condensed, or Extended.

convertFont:toSize: returns a font of the requested size, with all other characteristics the same as those of the original font.

convertWeight:ofFont: either increases or decreases the weight of the font provided, according to a boolean flag. Font weights are typically indicated by a series of names, which can vary from font to font. Some go from Light to Medium to Bold, while others have Book, SemiBold, Bold, and Black. This method offers a uniform way of incrementing and decrementing any font's weight.

The default implementation of font conversion is very conservative, making a change only if no other trait or aspect is affected. For example, if you try to convert Helvetica Oblique 12.0 point by adding the Bold trait, and only Helvetica Bold is available, the font isn't converted. You can create a subclass of NSFontManager and override the conversion methods to perform less conservative conversion, perhaps using Helvetica Bold in this case and losing the Oblique trait.

In addition to the font-conversion methods, NSFontManager defines **fontWithFamily:traits:weight:size:** to construct a font with a given set of characteristics. If you don't care to make a subclass of NSFontManager, you can use this method to perform approximate font conversions yourself.

Examining Fonts

In addition to converting fonts, NSFontManager provides information on which fonts are available to the application, and on the characteristics of any given font. **availableFonts** returns an array of the names of all fonts available. **availableFontNamesWithTraits:** filters the available fonts based on a font trait mask.

There are three methods for examining individual fonts. **fontNamed:hasTraits:** returns YES if the font matches the trait mask provided. **traitsOfFont:** returns a trait mask for a given font. **weightOfFont:** returns an approximate ranking of a font's weight on a scale of 0–15, where 0 is the lightest possible weight, 5 is Normal or Book weight, 9 is the equivalent of Bold, and 15 is the heaviest possible (often called Black or Ultra Black).

Customizing the Font Conversion System

If you need to customize the font conversion system by creating subclasses of NSFontManager or NSFontPanel, you must inform the NSFontManager class of this change with a **setFontManagerFactory:** or **setFontPanelFactory:** message, before either the shared font manager or shared font panel is created.

These methods record your class as the one to instantiate the first time the font manager or Font Panel is requested.

You may be able to avoid using subclasses if all you need is to add some custom controls to the Font Panel. In this case, you can invoke NSFontPanel's **setAccessoryView:** method to add an NSView below its font browser.

If you provide your own Font menu, you should register it with the font manager using the **setFontMenu:** method. The font manager is responsible for validating Font menu items and changing their titles and tags according to the selected font. For example, when the selected font is Italic the font manager changes the Italic Font menu item to Unitalic, and changes its tag to NSUnitalicFontMask. Your Font menu's items should use the appropriate action methods and tags. Here are some examples:

Font Menu Item	Action	Tag
Italic	**addFontTrait:**	NSItalicFontMask
Bold	**addFontTrait:**	NSBoldFontMask
Heavier	**modifyFont:**	NSHeavierFontAction
Larger	**modifyFont:**	NSSizeUpFontAction

Method Types

Getting the shared font manager

+ sharedFontManager

Changing the default font conversion classes

+ setFontManagerFactory:
+ setFontPanelFactory:

Getting available fonts

– availableFonts
– availableFontFamilies
– availableFontNamesWithTraits:

Setting and examining the selected font

– setSelectedFont:isMultiple:
– selectedFont
– isMultiple
– sendAction

Action methods

– addFontTrait:
– removeFontTrait:
– modifyFont:
– modifyFontViaPanel:

Converting fonts automatically

– convertFont:

Converting fonts manually

– convertFont:toFace:
– convertFont:toFamily:
– convertFont:toHaveTrait:
– convertFont:toNotHaveTrait:
– convertFont:toSize:
– convertWeight:ofFont:

Getting a particular font

– fontWithFamily:traits:weight:size:

Examining fonts

– traitsOfFont:
– fontNamed:hasTraits:
– weightOfFont:

Enabling the Font Panel and Font menu

– setEnabled:
– isEnabled

Setting the Font menu

– setFontMenu:
– fontMenu:

Getting the Font Panel

– fontPanel:
– orderFrontFontPanel:

Setting the delegate

– setDelegate:
– delegate

Setting the action method

– setAction:
– action

Class Methods

setFontManagerFactory:

+ (void)**setFontManagerFactory:**(Class)*aClass*

Sets the class object used to create the font manager to *aClass*, which should be a subclass of NSFontManager. When the NSFontManager class object receives a **sharedFontManager** message, it creates an instance of *aClass*, if no instance already exists. Your font manager class should implement **init** as its designated initializer. The default font manager factory is NSFontManager.

This method must be invoked before your application's main nib file is loaded, such as in the application delegate's **applicationWillFinishLaunching:** method.

See also: + **setFontPanelFactory:**

setFontPanelFactory:

+ (void)**setFontPanelFactory:**(Class)*factoryId*

Sets the class used to create the Font Panel to *aClass*, which should be a subclass of NSFontPanel. Invoke this method before accessing the Font Panel in any way, such as in the application delegate's **applicationWillFinishLaunching:** method.

See also: + **setFontManagerFactory:**

sharedFontManager

+ (NSFontManager *)**sharedFontManager**

Returns the singleton instance of the font manager factory for the application, creating it if necessary.

See also: + **setFontManagerFactory:**

Instance Methods

action

– (SEL)**action**

Returns the action that's sent to the first responder when the user selects a new font from the Font panel or chooses a command from the Font menu. The default action is **changeFont:**.

See also: – **setAction:**

addFontTrait:

– (void)**addFontTrait:**(id)*sender*

This action method causes the receiver to send its action message (**changeFont:** by default) up the responder chain. When a responder replies by providing a font to convert in a **convertFont:** message, the receiver converts the font by adding the trait specified by *sender*. This trait is determined by sending a **tag** message to *sender* and interpreting it as a font trait mask for a **convertFont:toHaveTrait:** message.

See also: – **removeFontTrait:**, – **modifyFont:**, – **modifyFontViaPanel:**

availableFontFamilies

– (NSArray *)**availableFontFamilies**

Returns the names of the font families available in the system. These fonts are in various system font directories.

See also: – **availableFontNamesWithTraits:**, – **availableFonts**

availableFontNamesWithTraits:

– (NSArray *)**availableFontNamesWithTraits:**(NSFontTraitMask)*fontTraitMask*

Returns the names of the fonts available in the system whose traits are described exactly by *fontTraitMask* (not the NSFont objects themselves). These fonts are in various system font directories. You specify the desired traits by combining these font trait mask values using the C bitwise OR operator:

- NSBoldFontMask
- NSCompressedFontMask
- NSCondensedFontMask
- NSExpandedFontMask
- NSFixedPitchFontMask
- NSItalicFontMask
- NSNarrowFontMask
- NSNonStandardCharacterSetFontMask
- NSPosterFontMask
- NSSmallCapsFontMask
- NSUnboldFontMask
- NSUnitalicFontMask

These pairs of traits are mutually exclusive:

- NSCondensedFontMask and NSExpandedFontMask
- NSBoldFontMask and NSUnboldFontMask
- NSItalicFontMask and NSUnitalicFontMask

If *fontTraitMask* is zero, this method returns all fonts that are neither italic nor bold. This is the same result you'd get if *fontTraitMask* were **NSUnitalicFontMask | NSUnboldFontMask**.

See also: – **availableFontFamilies**, – **availableFonts**

availableFonts

– (NSArray *)**availableFonts**

Returns the names of the fonts available in the system (not the NSFont objects themselves). These fonts are in various system font directories.

See also: – **availableFontFamilies**, – **availableFontNamesWithTraits:**

availableMembersOfFontFamily:

– (NSArray *)**availableMembersOfFontFamily:**(NSString *)*family*;

Returns an NSArray with one entry for each available membor of a font family. *family* is the name of a font family, like one that **availableFontFamilies** returns.

Each entry of the returned NSArray is another NSArray with 4 members, as follows:

- 0. The PostScript font name, as a NSString
- 1. The part of the font name used in the font panel that's not the font name, as a NSString. This is not localized. For example **"Roman"**, **"Italic"**, or **"Bold"**.
- 2. The font's weight, as a NSNumber
- 3. The font's traits, as a NSNumber

The members of the family are arranged in the font-panel order (narrowest to widest, lightest to boldest, plain to italic).

For example, if you call **availableMembersOfFontFamily:@"Times"**, it might return an array like this:

```
(("Times-Roman", "Roman", 5, 4),
 ("Times-Italic", "Italic", 6, 5),
 ("Times-Bold", "Bold", 9, 2),
 ("Times-BoldItalic", "Bold Italic", 9, 3)
)
```

convertFont:

– (NSFont *)**convertFont:**(NSFont *)*aFont*

Converts *aFont* according to the object that initiated a font change, typically the Font Panel or Font menu. Returns the converted font, or *aFont* itself if the conversion isn't possible.

This method is invoked in response to a **changeFont:** message, which is itself initiated by an action message such as **addFontTrait:** or **modifyFontViaPanel:**. These initiating methods cause the font manager to query the sender for the action to take and the traits to change. See the class description for more information.

See also: – **convertFont:toFace:**, – **convertFont:toFamily:**, – **convertFont:toHaveTrait:**, – **convertFont:toNotHaveTrait:**, – **convertFont:toSize:**, – **convertWeight:ofFont:**

convertFont:toFace:

– (NSFont *)**convertFont:**(NSFont *)*aFont* **toFace:**(NSString *)*typeface*

Returns an NSFont whose traits are as similar as possible to those of *aFont* except for the typeface, which is changed to *typeface*. Returns *aFont* if it can't be converted. A typeface is a fully specified family-face name, such as Helvetica-BoldOblique or Times-Roman.

This method attempts to match the weight and posture of *aFont* as closely as possible. Italic is mapped to Oblique, for example. Weights are mapped based on an approximate numeric scale of 0–15.

See also: – **convertFont:toFamily:**, – **convertFont:toHaveTrait:**, – **convertFont:toNotHaveTrait:**, – **convertFont:toSize:**, – **convertWeight:ofFont:**, – **convertFont:**

convertFont:toFamily:

– (NSFont *)**convertFont:**(NSFont *)*aFont* **toFamily:**(NSString *)*family*

Returns an NSFont whose traits are as similar as possible to those of *aFont* except for the font family, which is changed to *family*. Returns *aFont* if it can't be converted. A family is a generic font name, such as Helvetica or Times.

This method attempts to match the weight and posture of *aFont* as closely as possible. Italic is mapped to Oblique, for example. Weights are mapped based on an approximate numeric scale of 0–15.

See also: – **convertFont:toFace:**, – **convertFont:toHaveTrait:**, – **convertFont:toNotHaveTrait:**, – **convertFont:toSize:**, – **convertWeight:ofFont:**, – **convertFont:**

convertFont:toHaveTrait:

– (NSFont *)**convertFont:**(NSFont *)*aFont* **toHaveTrait:**(NSFontTraitMask)*fontTrait*

Returns an NSFont whose traits are the same as those of *aFont* except for the traits, which are changed to include the single trait *fontTrait*, which may be any one of:

- NSBoldFontMask
- NSCompressedFontMask
- NSCondensedFontMask
- NSExpandedFontMask
- NSFixedPitchFontMask
- NSItalicFontMask
- NSNarrowFontMask
- NSNonStandardCharacterSetFontMask
- NSPosterFontMask
- NSSmallCapsFontMask
- NSUnboldFontMask
- NSUnitalicFontMask

These pairs of traits are mutually exclusive:

- NSCondensedFontMask and NSExpandedFontMask
- NSBoldFontMask and NSUnboldFontMask
- NSItalicFontMask and NSUnitalicFontMask

Using NSUnboldFontMask or NSUnitalicFontMask removes the bold or italic trait, respectively.

Returns *aFont* if it can't be converted.

See also: – **convertFont:toNotHaveTrait:**, – **convertFont:toFace:**, – **convertFont:toFamily:**, – **convertFont:toSize:**, – **convertWeight:ofFont:**, – **convertFont:**

convertFont:toNotHaveTrait:

– (NSFont *)**convertFont:**(NSFont *)*aFont* **toNotHaveTrait:**(NSFontTraitMask)*fontTraitMask*

Returns an NSFont whose traits are the same as those of *aFont* except for the traits, which are changed so as not to include the single trait *fontTrait*, which may be any one of:

- NSBoldFontMask
- NSCompressedFontMask
- NSCondensedFontMask
- NSExpandedFontMask
- NSFixedPitchFontMask
- NSItalicFontMask
- NSNarrowFontMask

- NSNonStandardCharacterSetFontMask
- NSPosterFontMask
- NSSmallCapsFontMask
- NSUnboldFontMask
- NSUnitalicFontMask

These pairs of traits are mutually exclusive:

- NSCondensedFontMask and NSExpandedFontMask
- NSBoldFontMask and NSUnboldFontMask
- NSItalicFontMask and NSUnitalicFontMask

Using NSUnboldFontMask or NSUnitalicFontMask includes the bold or italic trait, respectively.

Returns *aFont* if it can't be converted.

See also: – **convertFont:toHaveTrait:**, – **convertFont:toFace:**, – **convertFont:toFamily:**, – **convertFont:toSize:**, – **convertWeight:ofFont:**, – **convertFont:**

convertFont:toSize:

– (NSFont *)**convertFont:**(NSFont *)*aFont* **toSize:**(float)*size*

Returns an NSFont whose traits are the same as those of *aFont* except for the size, which is changed to *size*. Returns *aFont* if it can't be converted.

See also: – **convertFont:toFace:**, – **convertFont:toFamily:**, – **convertFont:toHaveTrait:**, – **convertFont:toNotHaveTrait:**, – **convertWeight:ofFont:**, – **convertFont:**

convertWeight:ofFont:

– (NSFont *)**convertWeight:**(BOOL)*increaseFlag* **ofFont:**(NSFont *)*aFont*

Returns an NSFont whose weight is greater or lesser than that of *aFont*, if possible. If *increaseFlag* is YES, a heavier font is returned; if it's NO, a lighter font is returned. Returns *aFont* unchanged if it can't be converted.

Weights are graded along the following scale. The list on the right gives OpenStep's terminology and the list on the right gives the ISO equivalents. Names in the same line are treated as identical:

1) ultralight	
2) thin	W1) ultralight
3) light, extralight	W2) extralight

4) book	W3) light
5) regular,plain,display,roman	W4) semilight
6) medium	W5) medium
7) demi, demibold	
8) semi, semibold	W6) semibold
9) bold	W7) bold
10) extra, extrabold	W8) extrabold
11) heavy, heavyface	
12) black, super	W9) ultrabold
13) ultra, ultrablack, fat	
14) extrablack, obese, nord	

NSFontManager's implementation of this method refuses to convert a font's weight if it can't maintain all other traits, such as Italic and Condensed. You might wish to override this method to allow a looser interpretation of weight conversion.

See also: – **convertFont:toFace:**, – **convertFont:toFamily:**, – **convertFont:toHaveTrait:**, – **convertFont:toNotHaveTrait:**, – **convertFont:toSize:**, – **convertFont:**

delegate

– (id)**delegate**

Returns the receiver's delegate.

See also: – **setDelegate:**

fontMenu:

– (NSMenu *)**fontMenu:**(BOOL)*createFlag*

Returns the menu that's hooked up to the font conversion system, creating it if necessary and if *createFlag* is YES.

See also: – **setFontMenu:**

fontNamed:hasTraits:

– (BOOL)**fontNamed:**(NSString *)*typeface* **hasTraits:**(NSFontTraitMask)*fontTraitMask*

Returns YES if the font named *typeface* has all the traits specified in *fontTraitMask*, NO if it doesn't. You specify the desired traits by combining these font trait mask values using the C bitwise OR operator:

- NSBoldFontMask
- NSCompressedFontMask
- NSCondensedFontMask
- NSExpandedFontMask
- NSFixedPitchFontMask
- NSItalicFontMask
- NSNarrowFontMask
- NSNonStandardCharacterSetFontMask
- NSPosterFontMask
- NSSmallCapsFontMask
- NSUnboldFontMask
- NSUnitalicFontMask

These pairs of traits are mutually exclusive:

- NSCondensedFontMask and NSExpandedFontMask
- NSBoldFontMask and NSUnboldFontMask
- NSItalicFontMask and NSUnitalicFontMask

Using NSUnboldFontMask returns YES if the font is not bold; NO, otherwise. Using NSUnitalicFontMask returns YES if the font is not italic; NO, otherwise.

fontPanel:

– (NSFontPanel *)**fontPanel:**(BOOL)*createFlag*

Returns the application's shared Font Panel object, creating if necessary and if *createFlag* is YES.

See also: + **sharedFontPanel** (NSFontPanel), + **sharedFontPanelExists** (NSFontPanel), + **setFontPanelFactory:**

fontWithFamily:traits:weight:size:

– (NSFont *)**fontWithFamily:**(NSString *)*family*
traits:(NSFontTraitMask)*fontTraitMask*
weight:(int)*weight*
size:(float)*size*

Attempts to load a font with the specified characteristics, returning the font if successful and **nil** if not. *family* is the generic name of the font desired, such as Times or Helvetica. *weight* is a hint for the weight desired, on a scale of 0–15: a value of 5 indicates a normal or book weight and 9 or more a bold or heavier weight. The *weight* is ignored if *fontTraitMask* includes NSBoldFontMask.

You specify *fontTraitMask* by combining these font trait mask values using the C bitwise OR operator:

- NSBoldFontMask
- NSCompressedFontMask
- NSCondensedFontMask
- NSExpandedFontMask
- NSFixedPitchFontMask
- NSItalicFontMask
- NSNarrowFontMask
- NSNonStandardCharacterSetFontMask
- NSPosterFontMask
- NSSmallCapsFontMask
- NSUnboldFontMask
- NSUnitalicFontMask

These pairs of traits are mutually exclusive:

- NSCondensedFontMask and NSExpandedFontMask
- NSBoldFontMask and NSUnboldFontMask
- NSItalicFontMask and NSUnitalicFontMask

Using NSUnboldFontMask or NSUnitalicFontMask loads a font that doesn't have either the bold or italic trait, respectively.

isEnabled

– (BOOL)**isEnabled**

Returns YES if the font-conversion system's user interface items (the Font Panel and Font menu items) are enabled, NO if they're not.

See also: – **isEnabled** (NSFontPanel), – **isEnabled** (NSMenuItem), – **setEnabled:**

isMultiple

– (BOOL)**isMultiple**

Returns YES if the last font selection recorded has multiple fonts, NO if it's a single font.

See also: – **setSelectedFont:isMultiple:**, – **selectedFont**

localizedNameForFamily:face:

– (NSString *) **localizedNameForFamily:**(NSString *)*family* **face:**(NSString *)*face*;

Returns a localized string with the name of the specified font family and face (for example, **@"Times"** and **@"Roman"**), if one exists. The user's location is determined from the user's NSLanguages default setting. The method also loads the localized strings for the font, if they aren't already loaded.

If *face* is **nil**, this method returns the font family only.

modifyFont:

– (void)**modifyFont:**(id)*sender*

This action method causes the receiver to send its action message (**changeFont:** by default) up the responder chain. When a responder replies by providing a font to convert in a **convertFont:** message, the receiver converts the font in the manner specified by *sender*. The conversion is determined by sending a **tag** message to *sender* and invoking a corresponding method:

Sender's Tag	Method Used
NSNoFontChangeAction	None, the font is returned unchanged
NSViaPanelFontAction	The Font Panel's **panelConvertFont:**
NSAddTraitFontAction	**convertFont:toHaveTrait:**
NSRemoveTraitFontAction	**convertFont:toNotHaveTrait:**
NSSizeUpFontAction	**convertFont:toSize:**
NSSizeDownFontAction	**convertFont:toSize:**
NSHeavierFontAction	**convertWeight:ofFont:**
NSLighterFontAction	**convertWeight:ofFont:**

See also: – **addFontTrait:**, – **removeFontTrait:**, – **modifyFontViaPanel:**

modifyFontViaPanel:

– (void)**modifyFontViaPanel:**(id)*sender*

This action method causes the receiver to send its action message (**changeFont:** by default) up the responder chain. When a responder replies by providing a font to convert in a **convertFont:** message, the receiver converts the font by sending a **panelConvertFont:** message to the Font Panel. The panel in turn may send **convertFont:toFamily:**, **convertFont:toHaveTrait:**, and other specific conversion methods to make its change.

See also: – **addFontTrait:**, – **removeFontTrait:**, – **modifyFont:**

orderFrontFontPanel:

– (void)**orderFrontFontPanel:**(id)*sender*

This action method opens the Font Panel by sending it an **orderFront:** message, creating the Font Panel if necessary.

See also: – **fontPanel:**, + **setFontPanelFactory:**

removeFontTrait:

– (void)**removeFontTrait:**(id)*sender*

This action method causes the receiver to send its action message (**changeFont:** by default) up the responder chain. When a responder replies by providing a font to convert in a **convertFont:** message, the receiver converts the font by removing the trait specified by *sender*. This trait is determined by sending a **tag** message to *sender* and interpreting it as a font trait mask for a **convertFont:toNotHaveTrait:** message.

See also: – **addFontTrait:**, – **modifyFont:**, – **modifyFontViaPanel:**

selectedFont

– (NSFont *)**selectedFont**

Returns the last font recorded with a **setSelectedFont:isMultiple:** message. While fonts are being converted in response to a **changeFont:** message, you can determine the font selected in the Font Panel like this:

```
NSFontManager *fontManager = [NSFontManager sharedFontManager];

panelFont = [fontManager convertFont:[fontManager selectedFont]];
```

See also: – **isMultiple**

sendAction

– (BOOL)**sendAction**

Sends the receiver's action message, **changeFont:** by default, up the responder chain, initiating a font change for whatever conversion and trait to change were last requested. Returns YES if some object handled the **changeFont:** message, NO if the message went unheard.

This method is used internally by the font conversion system. You should never need to invoke it directly. Instead, use the action methods such as **addFontTrait:** or **modifyFontViaPanel:**.

See also: – **setAction:**

setAction:

– (void)**setAction:**(SEL)*aSelector*

Sets the action that's sent to the first responder when the user selects a new font from the Font panel or chooses a command from the Font menu to *aSelector*. The default action is **changeFont:**. You should rarely need to change this.

See also: – **action**

setDelegate:

– (void)**setDelegate:**(id)*anObject*

Sets the receiver's delegate to *anObject*.

See also: – **delegate**

setEnabled:

– (void)**setEnabled:**(BOOL)*flag*

Controls whether the font-conversion system's user interface items (the Font Panel and Font menu items) are enabled. If *flag* is YES they're enabled; if *flag* is NO they're disabled.

See also: – **setEnabled:** (NSFontPanel), – **isEnabled**

setFontMenu:

– (void)**setFontMenu:**(NSMenu *)*aMenu*

Records *aMenu* as the application's Font menu.

See also: – **fontMenu:**

setSelectedFont:isMultiple:

– (void)**setSelectedFont:**(NSFont *)*aFont* **isMultiple:**(BOOL)*flag*

Records *aFont* as the currently selected font, and updates the Font Panel to reflect this. If *flag* is YES, the Font Panel indicates that more than one font is contained in the selection.

An object that manipulates fonts should invoke this method whenever it becomes first responder and whenever its selection changes. It shouldn't invoke this method in the process of handling a **changeFont:** message, as this causes the font manager to lose the information necessary to effect the change. After all fonts have been converted, the font manager itself records the new selected font.

See also: – **selectedFont**, – **isMultiple**

traitsOfFont:

– (NSFontTraitMask)**traitsOfFont:**(NSFont *)*aFont*

Returns the traits of *aFont*, a mask created by combining these options with the C bitwise OR operator:

- NSBoldFontMask
- NSCompressedFontMask
- NSCondensedFontMask
- NSExpandedFontMask
- NSFixedPitchFontMask
- NSItalicFontMask
- NSNarrowFontMask
- NSNonStandardCharacterSetFontMask
- NSPosterFontMask
- NSSmallCapsFontMask

NSCondensedFontMask and NSExpandedFontMask are mutually exclusive.

weightOfFont:

– (int)**weightOfFont:**(NSFont *)*aFont*

Returns a rough numeric measure of *aFont*'s weight, where 0 indicates the lightest possible weight, 5 indicates a normal or book weight, and 9 or more indicates a bold or heavier weight.

Methods Implemented by Responders

– (void)**changeFont:**(id)*sender*

Informs responders of a font change: The user changed the font either in the selection of a rich text field or in a whole plain text field. Any object that contains a font which the user can change must respond to the **changeFont:** message by sending a **convertFont:** message back to sender (an NSFontManager object) for each font in the selection. For more information, see "Responding to Font Changes".

See also: – **addFontTrait:**, – **convertFont:toHaveTrait:**, – **convertFont:toFace:**, – **convertFont: toFamily:**, – **convertFont:toNotHaveTrait:**, – **convertFont:toSize:**, – **convertWeight:ofFont:**, – **convertFont:**, – **removeFontTrait:**, – **modifyFontViaPanel:**, – **modifyFont:**

Methods Implemented By the Delegate

fontManager:willIncludeFont:

– (BOOL)**fontManager:**(id)*theFontManager* **willIncludeFont:**(NSString *)*fontName*

Requests permission from the delegate to display *fontName* in the Font Panel. *fontName* is the full PostScript name of the font, such as "Helvetica-BoldOblique" or "Helvetica-Narrow-Bold". If the delegate returns YES, *fontName* is listed; if the delegate returns NO, it isn't.

This method is invoked repeatedly as necessary whenever the Font Panel needs updating, such as when the Font Panel is first loaded, and when the user selects a family name to see which typefaces in that family are available. Your implementation should execute fairly quickly to guarantee the responsiveness of the Font Panel.

NSFontPanel

Inherits From: NSPanel : NSWindow : NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSFontPanel.h

Class Description

The NSFontPanel class implements the Font Panel—a user-interface object that displays a list of available fonts, letting the user preview them and change the font used to display text. The actual changes are made through conversion messages sent to the shared NSFontManager instance. There's only one Font Panel for each application.

In general, you add the facilities of the Font Panel to your application, along with the NSFontManager and the Font menu, through Interface Builder. You do this by dragging a Font menu into one of your application's menus. At run time, when the user chooses the Font Panel command for the first time, the Font Panel object is created and hooked into the font conversion system. You can also create (or access) the Font Panel using the **sharedFontPanel** class method.

You can add a custom view object to an NSFontPanel using **setAccessoryView:**, or limit the fonts display by assigning a delegate to the application's font manager object. If you want the NSFontManager to instantiate the Font Panel from some class other than NSFontPanel, use NSFontManager's **setFontPanelFactory:** class method. See the NSFontManager class specification for more information on using the font conversion system.

Normally, the application's Font Panel displays all the standard fonts available on the system. If this isn't appropriate for your application—for example, if only fixed-pitch fonts should be used—you can assign a delegate to the NSFontPanel object to filter the available fonts. Before the NSFontPanel adds a particular font family or face to its list, it asks its delegate to confirm the addition by sending the delegate a **fontManager:willIncludeFont:** message. If the delegate returns YES (or doesn't implement this method), the font is added. If the delegate returns NO, the font isn't added.

Method Types

Getting the Font Panel

+ sharedFontPanel
+ sharedFontPanelExists

Enabling font changes

– setEnabled:
– isEnabled

Updating the Font Panel

– setPanelFont:isMultiple:

Converting fonts

– panelConvertFont:

Working in modal loops

– worksWhenModal

Setting an accessory view

– setAccessoryView:
– accessoryView

Class Methods

sharedFontPanel

+ (NSFontPanel *)**sharedFontPanel**

Returns the single NSFontPanel instance for the application, creating it if necessary.

See also: + **sharedFontPanelExists**, + **setFontPanelFactory:** (NSFontManager)

sharedFontPanelExists

+ (BOOL)**sharedFontPanelExists**

Returns YES if the shared Font Panel has been created, NO if it hasn't.

See also: + **sharedFontPanel**

Instance Methods

accessoryView

– (NSView *)**accessoryView**

Returns the receiver's accessory view.

See also: – **setAccessoryView:**

isEnabled

– (BOOL)**isEnabled**

Returns YES if the receiver's Set button is enabled, NO if it isn't. The receiver continues to reflect the font of the selection for cooperating text objects regardless of this setting.

See also: – **setEnabled:**

panelConvertFont:

– (NSFont *)**panelConvertFont:**(NSFont *)*aFont*

Converts *aFont* using the settings in the receiver, with the aid of the shared NSFontManager if necessary, and returns the converted font. If *aFont* can't be converted it's returned unchanged.

For example, if *aFont* is Helvetica Oblique 12.0 point and the user has selected the Times font family (and nothing else) in the Font Panel, the font returned is Times Italic 12.0 point.

See also: – **convertFont:** (NSFontManager)

setAccessoryView:

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

Establishes *aView* as the receiver's accessory view, allowing you to add custom controls to your application's Font Panel without having to create a subclass.

See also: – **accessoryView**

setEnabled:

– (void)**setEnabled:**(BOOL)*flag*

Controls whether the receiver's Set button is enabled. If *flag* is YES the Set button is enabled; if *flag* is NO it's disabled. The receiver continues to reflect the font of the selection for cooperating text objects regardless of this setting.

See also: – **isEnabled**

setPanelFont:isMultiple:

– (void)**setPanelFont:**(NSFont *)*aFont* **isMultiple:**(BOOL)*flag*

Sets the selected font in the receiver to *aFont* if *flag* is NO, otherwise selects no font and displays a message in the preview area indicating that multiple fonts are selected. You normally don't use this method directly;

instead, you send **setSelectedFont:isMultiple:** to the shared NSFontManager, which in turn invokes this method.

worksWhenModal

– (BOOL)**worksWhenModal**

Returns YES, regardless of the setting established using the NSPanel method **setWorksWhenModal:**. This allows fonts to be changed in modal windows and panels.

See also: – **worksWhenModal** (NSWindow), **– worksWhenModal** (NSPanel)

NSForm

Inherits From: NSMatrix : NSControl : NSView : NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSForm.h

Class Description

An NSForm is a vertical NSMatrix of NSFormCells. Here's an example:

In NSForm's methods, each NSFormCell is called an "entry" (or, sometimes, a "cell" or "item"). The left part of each entry is called the "title," and the right part is called the "text." Methods that refer to individual entries use an one-dimensional "index"; the indexing system starts at the top of the top of form, with zero.

Any entry in the form can be "selected." When an entry is selected, its text area responds to the user's keystrokes. You can select an entry using the **selectTextAtIndex:** method, or you can let the user select an entry by clicking it with the mouse. Once an entry is selected, the user can select the next entry by pressing Tab, or select the previous entry by pressing Shift-Tab.

To initiate the action of a selected entry, the user presses Return or Enter. In response, the entry sends an action message to its target. If the entry has no target, the NSForm sends an action message to *its* target.

NSForm includes a methods to change the appearance of entries (the **set...** methods). These methods affect every entry in the form. To change the appearance of an individual entry, you need to single it out, using **cellAtIndex:**, and then send it messages appropriate to an NSFormCell.

For more information, see the class specifications for NSFormCell and NSMatrix.

Method Types

Adding and removing entries

– addEntry:
– insertEntry:atIndex:
– removeEntryAtIndex:

Changing the appearance of all the entries

– setBezeled:
– setBordered:
– setEntryWidth:
– setInterlineSpacing:
– setTitleAlignment:
– setTextAlignment:
– setTitleFont:
– setTextFont:

Getting cells and indices

– indexOfCellWithTag:
– indexOfSelectedItem
– cellAtIndex:

Displaying a cell

– drawCellAtIndex:

Editing text

– selectTextAtIndex:

Instance Methods

addEntry:

– (NSFormCell *)**addEntry:**(NSString *)*title*

Adds a new entry to the end of the form, and gives it the title *title*. The new entry has no tag, target, or action, but is enabled and editable.

See also: – **insertEntry:atIndex:**, – **setTag:** (NSActionCell), – **setTarget:** (NSActionCell), – **setAction:** (NSActionCell), – **setEnabled:** (NSActionCell), – **setEditable:** (NSCell)

cellAtIndex:

– (id)**cellAtIndex:**(int)*entryIndex*

Returns the entry specified by *entryIndex*.

See also: – **indexOfCellWithTag:**, – **indexOfSelectedItem**

drawCellAtIndex:

– (void)**drawCellAtIndex:**(int)*entryIndex*

Displays the entry specified by *entryIndex*. Since this method is called automatically whenever a cell needs drawing, you will never need to invoke it explicitly. It is only included in the API so that you can override it if you subclass NSFormCell.

See also: – **indexOfCellWithTag:**, – **indexOfSelectedItem**

indexOfCellWithTag:

– (int)**indexOfCellWithTag:**(int)*tag*

Returns the index of the entry whose tag is *tag*.

See also: – **tag** (NSCell)

indexOfSelectedItem

– (int)**indexOfSelectedItem**

Returns the index of the selected entry. If no entry is selected, **indexOfSelectedItem** returns -1.

insertEntry:atIndex:

– (NSFormCell *)**insertEntry:**(NSString *)*title* **atIndex:**(int)*entryIndex*

Inserts an entry with the title *title* at the position in the form specified by *entryIndex*. The new entry has no tag, target, or action, and, as explained in the class description, it won't appear on the screen automatically.

Returns the newly inserted NSFormCell.

See also: – **addEntry:**, – **removeEntryAtIndex:**

removeEntryAtIndex:

– (void)**removeEntryAtIndex:**(int)*entryIndex*

If *entryIndex* is a valid position in the form, removes the entry there and frees it.

selectTextAtIndex:

– (void)**selectTextAtIndex:**(int)*entryIndex*

If *entryIndex* is a valid position in the Form, selects the entry at that position.

setBezeled:

– (void)**setBezeled:**(BOOL)*flag*

If *flag* is YES, sets all the entries in the form to show a bezel around their editable text; if *flag* is NO, sets all the entries to show no bezel.

See also: – **isBezeled** (Cell), – **setBordered:**

setBordered:

– (void)**setBordered:**(BOOL)*flag*

flag determines whether the entries in the form are set to display a border—that is, a thin line—around their editable text fields. An entry can have a border or a bezel, but not both.

See also: – **isBordered** (Cell), – **setBezeled:**

setEntryWidth:

– (void)**setEntryWidth:**(float)*width*

Sets the width (in pixels) of all the entries in the form. This width includes both the title and the text field.

setInterlineSpacing:

– (void)**setInterlineSpacing:**(float)*spacing*

Sets the number of pixels between entries in the form to *spacing*.

setTextAlignment:

– (void)**setTextAlignment:**(int)*alignment*

Sets the alignment for all of the form's editable text. *alignment* can be one of three constants: NSRightTextAlignment, NSCenterTextAlignment, or NSLeftTextAlignment (the default).

See also: – **setTitleAlignment:**

setTextFont:

– (void)**setTextFont:**(NSFont *)*font*

Sets the font for all of the form's editable text fields.

See also: – **setTitleFont:**

setTitleAlignment:

– (void)**setTitleAlignment:**(NSTextAlignment)*alignment*

Sets the alignment for all of the entry titles. *alignment* can be one of three constants: NSRightTextAlignment, NSCenterTextAlignment, or the default, NSLeftTextAlignment.

See also: – **setTextAlignment:**

setTitleFont:

– (void)**setTitleFont:**(NSFont *)*font*

Sets the font for all of the entry titles.

See also: – **setTextFont:**

NSFormCell

Inherits From: NSActionCell : NSCell : NSObject

Conforms To: NSCoding (from NSCell)
NSCopying (from NSCell)
NSObject (from NSObject)

Declared In: AppKit/NSFormCell.h

Class Description

This class is used to implement text entry fields in an NSForm. The left part of an NSFormCell is a title. The right part is an editable text entry field.

For more on the use of NSFormCell, see the class specification for NSForm.

Method Types

Initializing an NSFormCell
– initTextCell:

Asking about a cell's appearance
– isOpaque

Asking about a cell's title
– attributedTitle
– title
– titleAlignment
– titleFont
– titleWidth

Changing the cell's title
– setAttributedTitle:
– setTitle:
– setTitleAlignment:
– setTitleFont:
– setTitleWidth:

Setting a keyboard equivalent
– setTitleWithMnemonic:

Instance Methods

attributedTitle

– (NSAttributedString *)**attributedTitle**

Returns the title as an attributed string.

initTextCell:

– (id)**initTextCell:**(NSString *)*aString*

Initializes a newly allocated NSFormCell. Its title is set to *aString*; the contents of its text entry field are set to the empty string (“”). The font for both title and text is the user’s chosen system font in 12.0 point, and the text area is drawn with a bezel. This method is the designated initializer for NSFormCell.

Returns self.

See also: – **setTitle:**

isOpaque

– (BOOL)**isOpaque**

Returns YES if both the title and the entry field are opaque, NO if one or both of them are transparent. Since titles are transparent by default, this method usually returns NO.

setAttributedTitle:

– (void)**setAttributedTitle:**(NSAttributedString *)*anAttributedString*

Sets the cell’s title and title attributes according to *anAttributedString*.

setTitle:

– (void)**setTitle:**(NSString *)*aString*

Sets the cell’s title to *aString.*

setTitleAlignment:

– (void)**setTitleAlignment:**(NSTextAlignment)*alignment*

Sets the alignment of the title. *alignment* can be one of three constants: NSLeftTextAlignment, NSRightTextAlignment, or NSCenterTextAlignment.

setTitleFont:

– (void)**setTitleFont:**(NSFont *)*font*

Sets the title's font.

setTitleWidth:

– (void)**setTitleWidth:**(float)*width*

You usually won't need to to invoke this method, since the Application Kit automatically sets the title width whenever the title changes. If, however, the automatic width doesn't suit your needs, you can use **setTitleWidth:** to set the width in pixels.

Once you have set the width explicity this way, the Application Kit stops setting the width automatically; you will need to invoke **setTitleWidth:** every time the title changes. If you want the Application Kit to resume automatic width assignments, invoke **setTitleWidth:** with a negative *width* value.

setTitleWithMnemonic:

– (void)**setTitleWithMnemonic:**(NSString *)*titleWithAmpersand*

Sets the cell title and a single mnemonic character. The mnemonic character, which follows the ampersand in *titleWithAmpersand*, serves as an Alt-key equivalent to clicking in the text entry field.

For example, if *titleWithAmpersand* is "T&itle," the cell's title will be displayed as "Title" (the mnemonic character, *i*, is underlined). If a user types Alt-i, it will have the same effect as clicking in the text entry field.

See also: – **setTitle:**

title

– (NSString *)**title**

Returns the cell's title. The default title is "Field:".

titleAlignment

– (NSTextAlignment)**titleAlignment**

Returns the alignment of the title, which will be one of the following: NSLeftTextAlignment, NSCenterTextAlignment, or NSRightTextAlignment (the default).

titleFont

– (NSFont *)**titleFont**

Returns the font used to draw the cell's title.

titleWidth

– (float)**titleWidth**

Returns the width (in pixels) of the title field. If you specified the width using **setTitleWidth:**, this methdo returns the value you chose. Otherwise, it returns the width calculated automatically by the Application Kit.

See also: – **titleWidth:**

titleWidth:

– (float)**titleWidth:**(NSSize)*aSize*

Returns the width (in pixels) of the title field. If you specified the width using **setTitleWidth:**, this method returns the value you chose Otherwise, it calculates the width, constrained to *aSize*.

See also: – **titleWidth**

NSHelpManager

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: AppKit/NSHelpManager.h

Class Description

Note: *The Rhapsody Help system is currently under development, so the API of NSHelpManager may change significantly in future releases.*

NSHelpManager provides a platform-independent approach to displaying on-line help. An application contains one instance of NSHelpManager. Your application's code rarely needs to access NSHelpManager directly. Instead, you use Interface Builder and Project Builder to set up on-line help for your application.

OpenStep applications can run on multiple platforms, and each platform provides its own support for on-line help. It's important to users that applications use the native on-line help system (on Microsoft Windows, for instance, users want the Microsoft Windows help system and don't want to have to learn how to use a different help system), so NSHelpManager does not provide a comprehensive solution for presenting help. Instead, it provides cross-platform support for context-sensitive help and allows you to present more comprehensive help (conceptual and task-based help) in any way you choose.

Context Help

Context-sensitive help (also referred to as context help) gives the user a small amount of information when they help-click an interface item. For example, if the user help-clicks a menu item called "Copy," they should get context help that says something like "Copies the currently selected text to the pasteboard." This text appears in a small window near where the user help-clicked, and the window disappears when the user clicks anywhere else in the application.

Help-clicking is performed in any one of several ways, depending on the platform and the hardware used. On Mach platforms, help-clicking is performed when the user holds down the Help key, the F1 key, or the Alternate and Control keys while pressing the mouse button. On Microsoft Windows platforms, users press Shift-F1 and then press the mouse button to display context-sensitive help. Some Microsoft Windows applications also have a What's This menu item on the Help menu. When the user selects this item, the next mouse click displays context-sensitive help.

To provide context-sensitive help for your application, follow these steps:

1. For each interface item that needs context help, create an RTF or RTFD file containing the text and any images you want to appear when the user help-clicks that interface item. Try to keep the text as brief as possible and the images as small as possible.

The text that you write will appear in a small window just under the cursor when the user help-clicks an interface item. If the user help-clicks near the edge of the screen, text may appear off-screen. (This is especially prevalent when the user help-clicks a menu item on the Mach platform.) Use hard returns in your text so that the window will be as narrow as possible.

2. If you don't need to localize your context help files, in Project Builder simply add these files to your project under Context Help.

   If you do need to localize your context help files, first copy the files into the appropriate **.lproj** directory of your project, then add them to the project.

In Interface Builder, connect each interface item to its context help file by doing the following:

1. Bring up the Interface Builder inspector and choose the Help display. The Help display lists all the context help files associated with your application. (You may have to quit and restart Interface Builder to get this to occur.)
2. Select an interface item.
3. In the Inspector, choose the appropriate help file.

When you build your application, **/usr/bin/compileHelp** packages your help files into a property list named **Help.plist**. NSHelpManager knows how to extract context help from a **Help.plist** file.

Comprehensive Help

Most applications provide some form of on-line help that is more comprehensive and detailed than context-sensitive help, such as conceptual or task help. NSHelpManager allows you to provide this sort of comprehensive help in any way you choose. Some help authors prefer to provide comprehensive help in HTML using a World-Wide Web browser; others use tools such as Digital Librarian or Concurrence; on Microsoft Windows a full-featured native help system is available.

When the user chooses the Help menu item, the NSApplication method **showHelp:** is invoked. This method simply asks NSWorkspace to open the help file you have specified for your application. That file should be the starting point of your help, and should allow users to access whatever information they might need.

To specify a help file for your application, do one of the following:

- In Project Builder, specify the name of the help file in the Project Attributes inspector. (If you are creating an application that will run on both the Mach and Windows platforms, you need to enter this file twice—once for Mach and once for Windows). The specified value can be a full or relative path, and if it is relative, it is assumed to be a resource in the application wrapper.
- As an alternative, you can place the help file in your application wrapper and name it after your application. If you haven't specified a help file, NSHelpManager looks in the application wrapper for an appropriately named file.

  On Mach, it must be an RTF file called *appName***.rtf** (where *appName* is the name of the application).

On Microsoft Windows, it must be a Windows help file called *appName*.**hlp**.

Note: It's common for Windows applications to have more than one command under the Help menu and to have each command open a different help file. To implement this, connect each of the Help menu commands to a different action method. The action methods should send **openFile:** to the shared NSWorkspace object to open the appropriate help file. For example:

```
[[NSWorkspace sharedWorkspace] openFile:@"AppKit.hlp"];
```

Method Types

Creating an NSHelpManager instance
+ sharedHelpManager

Getting and setting context help mode
+ setContextHelpModeActive:
+ isContextHelpModeActive

Returning context-sensitive help
– contextHelpForObject:
– showContextHelpForObject:locationHint:

Setting up context-sensitive help
– setContextHelp:forObject:
– removeContextHelpForObject:

Class Methods

isContextHelpModeActive

+ (BOOL)**isContextHelpModeActive**

Returns YES if the application is currently in context-sensitive help mode, NO otherwise. In context-sensitive help mode, when a user clicks a user interface item, help for that item is displayed in a small window just below the cursor.

See also: + **setContextHelpModeActive:**

setContextHelpModeActive:

+ (void)**setContextHelpModeActive:**(BOOL)*flag*

Controls context-sensitive help mode. If *flag* is YES, the application enters context-sensitive help mode. If *flag* is NO, the application returns to normal operation.

You never send this message directly; instead, the NSApplication method **activateContextHelpMode:** activates context-sensitive help mode, and the first mouse click after displaying the context-sensitive help window deactivates it.

When the application enters context-sensitive help mode, NSHelpManager posts NSContextHelpModeDidActivateNotification to the default notification center. When the application returns to normal operation, NSHelpManager posts NSContextHelpModeDidDeactivateNotification.

See also: + **isContextHelpModeActive**

sharedHelpManager

+ (NSHelpManager *)**sharedHelpManager**

Returns the shared NSHelpManager instance, creating it if it does not already exist.

Instance Methods

contextHelpForObject:

– (NSAttributedString *)**contextHelpForObject:**(id)*object*

Returns context-sensitive help for *object*.

See also: – **setContextHelp:forObject:**, – **showContextHelpForObject:locationHint:**

removeContextHelpForObject:

– (void)**removeContextHelpForObject:**(id)*object*

Removes the association between *object* and its context-sensitive help. If object does not have context-sensitive help associated with it, this method does nothing. Typically, you use Interface Builder to remove context-sensitive help from an item.

See also: – **setContextHelp:forObject:**

setContextHelp:forObject:

– (void)**setContextHelp:**(NSAttributedString *)*help* **forObject:**(id)*object*

Associates *help* with *object*. When the application enters context-sensitive help mode, if *object* is clicked, *help* will appear in the context-sensitive help window. Typically, you use Interface Builder to associate context-sensitive help with an object.

See also: – **removeContextHelpForObject**:

showContextHelpForObject:locationHint:

– (BOOL)**showContextHelpForObject:**(id)*object* **locationHint:**(NSPoint)*point*

Displays the context-sensitive help for *object* at or near the point on the screen specified by *point*. This point is usually just under the cursor. Returns YES if it successfully displays context-sensitive help for the object, NO if it cannot (for example, if there is no context-sensitive help associated with this object).

See also: – **contextHelpForObject:**

Notifications

NSContextHelpModeDidActivateNotification

Posted when the application enters context-sensitive help mode. This typically happens when the user holds down the Help key. It can also occur on Microsoft Windows platforms if the user chooses the What's This command from the Help menu.

This notification contains a notification object but no userInfo dictionary. The notification object is the NSHelpManager object.

NSContextHelpModeDidDeactivateNotification

Posted when the application exits context-sensitive help mode. This happens when the user clicks the mouse anywhere on the screen after displaying a context-sensitive help topic.

This notification contains a notification object but no userInfo dictionary. The notification object is the NSHelpManager object.

NSImage

Inherits From:	NSObject
Conforms To:	NSCoding NSCopying NSObject (from NSObject)
Declared In:	AppKit/NSImage.h

Class Description

An NSImage object contains an image that can be composited anywhere without first being drawn in any particular view. It manages the image by:

- Reading image data from the application bundle, from an NSPasteboard, or from an NSData object.
- Keeping multiple representations of the same image.
- Choosing the representation that's appropriate for a particular data type.
- Choosing the representation that's appropriate for any given display device.
- Caching the representations it uses by rendering them in off-screen windows.
- Optionally retaining the data used to draw the representations, so that they can be reproduced when needed.
- Compositing the image from the off-screen cache to where it's needed on-screen.
- Reproducing the image for the printer so that it matches what's displayed on-screen, yet is the best representation possible for the printed page.
- Automatically using any filtering services installed by the user to convert image data from unsupported formats to supported formats.

Defining an Image

An image can be created from various types of data:

- Encapsulated PostScript code (EPS)
- Bitmap data in Tag Image File Format (TIFF)
- Bitmap data in Windows Bitmap format (BMP)
- Untagged (raw) bitmap data

- Other image data supported by an NSImageRep subclass registered with the NSImage class
- Data that can be filtered to a supported type by a user-installed filter service

If data is placed in a file (for example, in an application bundle), the NSImage object can access the data whenever it's needed to create the image. If data is read from an NSData object, the NSImage object may need to store the data itself.

Images can also be defined by the program, in two ways:

- By drawing the image in an off-screen window maintained by the NSImage object. In this case, the NSImage maintains only the cached image.
- By defining a method that can be used to draw the image when needed. This allows the NSImage to delegate responsibility for producing the image to some other object.

Image Representations

An NSImage object can keep more than one representation of an image. Multiple representations permit the image to be customized for the display device. For example, different hand-tuned TIFF images can be provided for monochrome and color screens, and an EPS representation or a custom method might be used for printing. All representations are versions of the same image.

An NSImage returns an NSArray of its representations in response to a **representations** message. Each representation is a kind of NSImageRep object:

NSEPSImageRep	An image that can be recreated from EPS data that's either stored by the object or at a known location in the file system.
NSBitmapImageRep	An image that can be recreated from bitmap or TIFF data.
NSCustomImageRep	An image that can be redrawn by a method defined in the application.
NSCachedImageRep	An image that has been rendered in an off-screen cache from data or instructions that are no longer available. The image in the cache provides the only data from which the image can be reproduced.

You can define other NSImageRep subclasses for objects that render images from other types of source data. To make these new subclasses available to an NSImage object, they need to be added to the NSImageRep class registry by invoking the **registerImageRepClass:** class method. NSImage determines the data types that each subclass can support by invoking its **imageUnfilteredFileTypes** and **imageUnfilteredPasteboardTypes** methods.

Choosing Representations

The NSImage object will choose the representation that best matches the rendering device. By default, the choice is made according to the following set of ordered rules. Each rule is applied in turn until the choice of representation is narrowed to one.

1. Choose a color representation for a color device, and a gray-scale representation for a monochrome device.
2. Choose a representation with a resolution that matches the resolution of the device, or if no representation matches, choose the one with the highest resolution.

   By default, any image representation with a resolution that's an integer multiple of the device resolution is considered to match. If more than one representation matches, the NSImage will choose the one that's closest to the device resolution. However, you can force resolution matches to be exact by passing NO to the **setMatchesOnMultipleResolution:** method.

   Rule 2 prefers TIFF and bitmap representations, which have a defined resolution, over EPS representations, which don't. However, you can use the **setUsesEPSOnResolutionMismatch:** method to have the NSImage choose an EPS representation in case a resolution match isn't possible.
3. If all else fails, choose the representation with a specified bits per sample that matches the depth of the device. If no representation matches, choose the one with the highest bits per sample.

By passing NO to the **setPrefersColorMatch:** method, you can have the NSImage try for a resolution match before a color match. This essentially inverts the first and second rules above.

If these rules fail to narrow the choice to a single representation—for example, if the NSImage has two color TIFF representations with the same resolution and depth—the one that will be chosen is system dependent.

Caching Representations

When first asked to composite the image, the NSImage object chooses the representation that's best for the destination display device, as outlined above. It renders the representation in an off-screen window on the same device, then composites it from this cache to the desired location. Subsequent requests to composite the image use the same cache. Representations aren't cached until they're needed for compositing.

When printing, the NSImage tries not to use the cached image. Instead, it attempts to render on the printer—using the appropriate image data, or a delegated method—the best version of the image that it can. Only as a last resort will it image the cached bitmap.

Image Size

Before an NSImage can be used, the size of the image must be set, in units of the base coordinate system. If a representation is smaller or larger than the specified size, it can be scaled to fit.

If the size of the image hasn't already been set when the NSImage is provided with a representation, the size will be set from the data. The bounding box is used to determine the size of an NSEPSImageRep. The TIFF fields "ImageLength" and "ImageWidth" are used to determine the size of an NSBitmapImageRep.

Coordinate Systems

Images have the horizontal orientation of the base coordinate system; they can't be rotated or horizontally flipped. When composited, an image maintains this orientation, no matter what coordinate system it's composited to. (The destination coordinate system is used only to determine the location of a composited image, not its size or orientation.) Images can be flipped in the vertical direction by using **setFlipped:**.

It's possible to refer to portions of an image when compositing by specifying a rectangle in the image's coordinate system, which is identical to the base coordinate system, except that the origin is at the lower left corner of the image.

Named Images

An NSImage object can be identified either by its **id** or by a name. Assigning an NSImage a name adds it to a table kept by the class object; each name in the database identifies one and only one instance of the class. When you ask for an NSImage object by name (with the **imageNamed:** method), the class object returns the one from its database, which also includes all the system bitmaps provided by the Application Kit. If there's no object in the database for the specified name, the class object tries to create one by checking for a system bitmap of the same name, checking the name of the application's own image, and then checking for the image in the application's main bundle.

If a file matches the name, an NSImage is created from the data stored there. You can therefore create NSImage objects simply by including EPS, BMP, or TIFF data for them within the executable file, or in files inside the application's file package.

Image Filtering Services

NSImage is designed to automatically take advantage of user-installed filter services for converting unsupported image file types to supported image file types. The class method **imageFileTypes** returns an array of all file types from which NSImage can create an instance of itself. This list includes all file types supported by registered subclasses of NSImageRep, and those types that can be converted to supported file types through a user-installed filter service.

Adopted Protocols

NSCoding

– encodeWithCoder:
– initWithCoder:

NSCopying

– copyWithZone:

Method Types

Initializing a new NSImage instance

– initByReferencingFile:
– initWithBitmapHandle:
– initWithContentsOfFile:
– initWithData:
– initWithIconHandle:
– initWithPasteboard:
– initWithSize:

Setting the size of the image

– setSize:
– size

Referring to images by name

+ imageNamed:
– setName:
– name

Specifying the image

– addRepresentation:
– addRepresentations:
– lockFocus
– lockFocusOnRepresentation:
– unlockFocus

Using the image

– compositeToPoint:operation:
– compositeToPoint:fromRect:operation:
– dissolveToPoint:fraction:
– dissolveToPoint:fromRect:fraction:

Choosing which image representation to use

– setPrefersColorMatch:
– prefersColorMatch
– setUsesEPSOnResolutionMismatch:
– usesEPSOnResolutionMismatch
– setMatchesOnMultipleResolution:
– matchesOnMultipleResolution

Getting the representations

– bestRepresentationForDevice:
– representations
– removeRepresentation:

Determining how the image is stored

– setCachedSeparately:
– isCachedSeparately
– setDataRetained:
– isDataRetained
– setCacheDepthMatchesImageDepth:
– cacheDepthMatchesImageDepth

Determining how the image is drawn

– isValid
– setScalesWhenResized:
– scalesWhenResized
– setBackgroundColor:
– backgroundColor
– setFlipped:
– isFlipped
– drawRepresentation:inRect:
– recache

Assigning a delegate

– setDelegate:
– delegate

Producing TIFF data for the image

– TIFFRepresentation
– TIFFRepresentationUsingCompression:factor:

Managing NSImageRep subclasses

+ imageUnfilteredFileTypes
+ imageUnfilteredPasteboardTypes

Testing image data sources

+ canInitWithPasteboard:
+ imageFileTypes
+ imagePasteboardTypes

Class Methods

canInitWithPasteboard:

+ (BOOL)**canInitWithPasteboard:**(NSPasteboard *)*pasteboard*

Tests whether the receiver can create an instance of itself from the data represented by pasteboard. Returns YES if the receiver's list of registered NSImageReps includes a class that can handle the data represented by pasteboard.

NSImage uses the NSImageRep class method **imageUnfilteredPasteboardTypes** to find a class that can handle the data in *pasteboard*. When creating a subclass of NSImageRep that accepts image data from a non-default pasteboard type, override the **imageUnfilteredPasteboardTypes** method to notify NSImage of the pasteboard types your class supports.

See also: **+ imagePasteboardTypes**

imageFileTypes

+ (NSArray *)**imageFileTypes**

Returns an array of strings representing those file types for which a registered NSImageRep exists. This list includes all file types supported by registered subclasses of NSImageRep, plus those types that can be converted to supported file types through a user-installed filter service. The array returned by this method may be passed directly to the NSOpenPanel's **runModalForTypes:** method.

File types are identified by extension. By default, the list returned by this method contains "tiff", "tif", "bmp", and "eps".

When creating a subclass of NSImageRep that accepts image data from non-default file types, override NSImageRep's **imageUnfilteredFileTypes** method to notify NSImage of the file types your class supports.

See also: + **imageUnfilteredFileTypes**

imageNamed:

+ (id)**imageNamed:**(NSString *)*name*

Returns the NSImage instance associated with *name*. The returned object can be:

- One that's been assigned a name with the **setName:** method
- One of the named system bitmaps provided by the Application Kit

If there's no known NSImage with *name*, this method tries to create one by searching for image data in the application's executable file and in the main bundle (see NSBundle's class description for a description of how the bundle's contents are searched). If a file contains data for more than one image, a separate representation is created for each. If an image representation can't be found for *name*, no object is created and **nil** is returned.

The preferred way to name an image is to ask for a *name* without the extension, but to include the extension for a file name.

One particularly useful image is referenced by the string "NSApplicationIcon". If you supply this string to **imageNamed:**, the returned image will be the application's own icon. Icons for other applications can be obtained through the use of methods declared in the NSWorkspace class.

The image returned by this method should not be freed, unless it's certain that no other objects reference it.

See also: – **setName:**, – **name**, **– iconForFile:** (NSWorkspace), **+ imageFileTypes**

imagePasteboardTypes

+ (NSArray *)**imagePasteboardTypes**

Returns a null-terminated list of pasteboard types for which a registered NSImageRep exists. This list includes all pasteboard types supported by registered subclasses of NSImageRep, and those that can be converted to supported pasteboard types through a user-installed filter service.

By default, the list returned by this method contains "NSPostScriptPboardType" and "NSTIFFPboardType".

When creating a subclass of NSImageRep that accepts image data from non-default pasteboard types, override NSImageRep's **imageUnfilteredPasteboardTypes** method to notify NSImage of the pasteboard types your class supports.

See also: + **imageUnfilteredPasteboardTypes**

imageUnfilteredFileTypes

+ (NSArray *)**imageUnfilteredFileTypes**

Returns a null-terminated array of strings representing those file types for which a registered NSImageRep exists. This list consists of all file types supported by registered subclasses of NSImageRep, and doesn't include those types that can be converted to supported file types through a user-installed filter service. The array returned by this method may be passed directly to the NSOpenPanel's **runModalForTypes:** method.

See also: + **imageFileTypes**

imageUnfilteredPasteboardTypes

+ (NSArray *)**imageUnfilteredPasteboardTypes**

Returns a null-terminated list of pasteboard types for which a registered NSImageRep exists. This list consists of all pasteboard types supported by registered subclasses of NSImageRep, and doesn't include those that can be converted to supported pasteboard types through a user-installed filter service.

See also: + **imagePasteboardTypes**

Instance Methods

addRepresentation:

– (void)**addRepresentation:**(NSImageRep *)*imageRep*

Adds *imageRep* to the receiver's list of representations. After invoking this method, you may need to explicitly set features of the new representation, such as size, number of colors, and so on. This is true in particular if the NSImage has multiple image representations to choose from. See NSImageRep and its subclasses for the methods you use to complete initialization.

Any representation that's added by this method is retained by the NSImage. Note that representations can't be shared among NSImages.

See also: **– representations**, **– removeRepresentation:**

addRepresentations:

– (void)**addRepresentations:**(NSArray *)*imageReps*

Adds each of the representations in *imageReps* to the receiver's list of representations. After invoking this method, you may need to explicitly set features of the new representations, such as size, number of colors, and so on. This is true in particular if the NSImage has multiple image representations to choose from. See NSImageRep and its subclasses for the methods you use to complete initialization.

Representations added by this method are retained by the NSImage. Note that representations can't be shared among NSImages.

See also: **– representations**, **– removeRepresentation:**

backgroundColor

– (NSColor *)**backgroundColor**

Returns the background color of the rectangle where the image is cached. If no background color has been specified, NSColor's **clearColor** is returned, indicating a transparent background.

The background color will be visible when the image is composited only if the image doesn't completely cover all the pixels within the area specified for its size.

bestRepresentationForDevice:

– (NSImageRep *)**bestRepresentationForDevice:**(NSDictionary *)*deviceDescription*

Returns the best representation for the device described by *deviceDescription*. If *deviceDescription* is **nil**, the current device is assumed. "Choosing Representations" in the class introduction outlines the process

NSImage goes through in order to determine the “best” representation for a given device. For a list of dictionary keys and values appropriate to display and print devices, see NSGraphics.h.

See also: **– representations**, **– prefersColorMatch**

cacheDepthMatchesImageDepth

– (BOOL)**cacheDepthMatchesImageDepth**

Returns NO if the application’s default depth limit applies to the off-screen windows where the NSImage’s representations are cached. If window depths are instead determined by the specifications of the representations, **cacheDepthMatchesImageDepth** returns YES.

compositeToPoint:fromRect:operation:

– (void)**compositeToPoint:**(NSPoint)*aPoint*
fromRect:(NSRect)*aRect*
operation:(NSCompositingOperation)*op*

Composites the portion of the image enclosed by the *aRect* rectangle to the location specified by *aPoint* in the current coordinate system. *aRect* must be a valid (non-null) rectangle. The *aPoint* argument is the same as for **compositeToPoint:operation:**. *op* should be one of the compositing operations as defined in **dpsOpenStep.h**.

The source rectangle is specified relative to a coordinate system that has its origin at the lower left corner of the image, but is otherwise the same as the base coordinate system.

This method doesn’t check to be sure that the rectangle encloses only portions of the image. Therefore, it can conceivably composite areas that don’t properly belong to the image, if the *aRect* rectangle happens to include them. If this turns out to be a problem, you can prevent it from happening by having the NSImage cache its representations in their own individual windows (with the **setCachedSeparately:** method). In this case, the window’s clipping path will prevent anything but the image from being composited.

Compositing part of an image is as efficient as compositing the whole image, but printing just part of an image is not. When printing, it’s necessary to draw the whole image and rely on a clipping path to be sure that only the desired portion appears.

See also: **– dissolveToPoint:fromRect:fraction:**

compositeToPoint:operation:

– (void)**compositeToPoint:**(NSPoint)*aPoint*
operation:(NSCompositingOperation)*op*

Composites the image to the location specified by *aPoint* using the specified compositing operation, *op*.

aPoint is specified in the current coordinate system—the coordinate system of the currently focused NSView—and designates where the lower left corner of the image will appear. The image will have the orientation of the base coordinate system, regardless of the destination coordinates. *op* should be one of the compositing operations as defined in **dpsOpenStep.h**.

The image is composited from its off-screen window cache. Since the cache isn't created until the image representation is first used, this method may need to render the image before compositing.

When printing, the compositing methods do not composite, but attempt to render the same image on the page that compositing would render on the screen, choosing the best available representation for the printer. The *op* argument is ignored.

See also: **– dissolveToPoint:fraction:**

delegate

– (id)**delegate**

Returns the delegate of the NSImage object, or **nil** if no delegate has been set.

dissolveToPoint:fraction:

– (void)**dissolveToPoint:**(NSPoint)*aPoint*
fraction:(float)*aFloat*

Composites the image to the location specified by *aPoint*, just as **compositeToPoint:operation:** does, but uses the **dissolve** operator rather than **composite**. *aFloat* is a fraction between 0.0 and 1.0 that specifies how much of the resulting composite will come from the NSImage. If the source image contains alpha, this operation may promote the destination NSWindow to contain alpha.

To slowly dissolve one image into another, this method (or **dissolveToPoint:fromRect:fraction:**) needs to be invoked repeatedly with an ever-increasing *aFloat*. Since *aFloat* refers to the fraction of the source image that's combined with the original destination (not the destination image after some of the source has been dissolved into it), the destination image should be replaced with the original destination before each invocation. This is best done in a buffered window before the results of the composite are flushed to the screen.

When printing, this method is identical to **compositeToPoint:operation:**. The *delta* argument is ignored.

dissolveToPoint:fromRect:fraction:

– (void)**dissolveToPoint:**(NSPoint)*aPoint*
fromRect:(NSRect)*aRect*
fraction:(float)*aFloat*

Composites the *aRect* portion of the image to the location specified by *aPoint*, just as **compositeToPoint: fromRect:operation:** does, but uses the **dissolve** operator rather than **composite**. *aFloat* is a fraction between 0.0 and 1.0 that specifies how much of the resulting composite will come from the NSImage. If the source image contains alpha, this operation may promote the destination NSWindow.

When printing, this method is identical to **compositeToPoint:fromRect:operation:**. The *aFloat* argument is ignored.

drawRepresentation:inRect:

– (BOOL)**drawRepresentation:**(NSImageRep *)*imageRep*
inRect:(NSRect)*rect*

Fills the specified rectangle with the background color, then sends the *imageRep* a **drawInRect:** message to draw itself inside the rectangle (if the NSImage is scalable), or a **drawAtPoint:** message to draw itself at the location of the rectangle (if the NSImage is not scalable). The rectangle is located in the current window and is specified in the current coordinate system. This method returns the value returned by the **drawInRect:** or **drawAtPoint:** method, which indicates whether or not the representation was successfully drawn.

This method shouldn't be called directly; the NSImage uses it to cache and print its representations. By overriding it in a subclass, you can change how representations appear in the cache, and thus how they'll appear when composited. For example, your version of the method could scale or rotate the coordinate system, then send a message to **super** to perform this version.

If the background color is fully transparent and the image isn't being cached by the NSImage, the rectangle won't be filled before the representation draws.

initByReferencingFile:

– (id)**initByReferencingFile:**(NSString *)*filename*

Initializes the receiver, a newly allocated NSImage instance, for the file *filename*. This method initializes lazily: the NSImage doesn't actually open *filename* or create image representations from its data until an application attempts to composite or requests information about the NSImage.

filename may be a full or relative pathname, and should include an extension that identifies the data type in the file. The mechanism that actually creates the image representation for *filename* will look for an NSImageRep subclass that handles that data type from among those registered with NSImage. By default, the files handled are those with the extensions "tiff", "tif", "bmp", and "eps".

After finishing the initialization, this method returns **self**. However, if the new instance can't be initialized, it's freed and **nil** is returned. Since this method doesn't actually create image representations for the data, your application should do error checking before attempting to use the image; one way to do so is by invoking the **isValid** method to check whether the image can be drawn.

This method invokes **setDataRetained:**YES, thus enabling it to hold onto its file name. Note that if an image created with this method is archived, only the file name will be saved.

initWithBitmapHandle:

– (id)**initWithBitmapHandle:**(void *)*bitmap*

On Microsoft Windows platforms, **initWithBitmapHandle:** initializes the receiver, a newly allocated NSImage instance, with the contents of the Windows bitmap indicated by *bitmap*. If **initWithBitmapHandle:** is able to create one or more image representations, it returns **self**. Otherwise, the receiver is freed and **nil** is returned.

initWithContentsOfFile:

– (id)**initWithContentsOfFile:**(NSString *)*filename*

Initializes the receiver, a newly allocated NSImage instance, with the contents of the file *filename*. Unlike **initByReferencingFile:**, this method opens *filename* and creates one or more image representations from its data.

filename may be a full or relative pathname, and should include an extension that identifies the data type in the file. **initWithContentsOfFile:** will look for an NSImageRep subclass that handles that data type from among those registered with NSImage. By default, the files handled are those with the extensions "tiff", "tif", "bmp", and "eps".

After finishing the initialization, this method returns **self**. However, if at least one image representation can't be created from the contents of the specified file, the receiver is freed and **nil** is returned.

initWithData:

– (id)**initWithData:**(NSData *)*data*

Initializes the receiver, a newly allocated NSImage instance, with the contents of the data object *data*. If **initWithData:** is able to create one or more image representations, it returns **self**. Otherwise, the receiver is freed and **nil** is returned.

initWithIconHandle:

– (id)**initWithIconHandle:**(void *)*icon*

On Microsoft Windows platforms, **initWithIconHandle:** initializes the receiver, a newly allocated NSImage instance, with the contents of the Windows icon indicated by *icon*. If **initWithIconHandle:** is able to create one or more image representations, it returns **self**. Otherwise, the receiver is freed and **nil** is returned.

initWithPasteboard:

– (id)**initWithPasteboard:**(NSPasteboard *)*pasteboard*

Initializes and returns the receiver, a newly allocated NSImage instance, from *pasteboard*. *pasteboard* should contain a type returned by one of the registered NSImageRep's **imageUnfilteredPasteboardTypes** methods; the default types supported are NSPostscriptPboardType (NSEPSImageRep) and NSTIFFPboardType (NSBitmapImageRep). If *pasteboard* contains an NSFilenamesPboardType, the file name should have an extension returned by one of the registered NSImageRep's **imageUnfilteredFileTypes** methods; the default types supported are "tiff", "tif", "bmp", (all in NSBitmapImageRep) and "eps" (NSEPSImageRep).

If **initWithPasteboard:** is able to create one or more image representations, it returns **self**. Otherwise, the receiver is freed and **nil** is returned.

initWithSize:

– (id)**initWithSize:**(NSSize)*aSize*

Initializes the receiver, a newly allocated NSImage instance, to *aSize* and returns **self**. The size should be specified in units of the base coordinate system. Although you can initialize the receiver without specifying a size by passing a size of (0.0, 0.0) to **initWithSize:**, the receiver's size must be set before the NSImage can be used.

See also: **– setSize:**

isCachedSeparately

– (BOOL)**isCachedSeparately**

Returns YES if each representation of the image is cached separately in an off-screen window of its own, and NO if they can be cached in off-screen windows together with other images. A return of NO doesn't mean that the windows are, in fact, shared, just that they can be. The default is NO.

isDataRetained

– (BOOL)**isDataRetained**

Returns YES if the NSImage retains the data needed to render the image, and NO if it doesn't. The default is NO, except for images created with **initByReferencingFile:**, which should hold onto their file names. If the data is available in a file that won't be moved or deleted, or if responsibility for drawing the image is delegated to another object with a custom method, there's no reason for the NSImage to retain the data. However, if the NSImage reads image data from a file created with **initWithContentsOfFile:**, you may want to have it keep the data itself; for example, to render the same image on another device at a different resolution.

isFlipped

– (BOOL)**isFlipped**

Returns YES if a vertically flipped coordinate system is used when locating the image, and NO if it isn't. The default is NO.

isValid

– (BOOL)**isValid**

Returns YES if a representation for the receiver can drawn in the cache, and NO if it can't; for example, because the file from which it was initialized is non-existent, or the data in that file is invalid.

If no representations exist for the receiver, **isValid** first creates a cache with the default depth.

See also: **– initByReferencingFile:**

lockFocus

– (void)**lockFocus**

Prepares for drawing of the best representation of the NSImage for the current device by making the off-screen window where the representation will be cached the current window and a coordinate system specific to the area where the image will be drawn the current coordinate system. If the receiver has no representations, **lockFocus** first creates one with the default depth. See "Choosing Representations" in the class description for information on how the "best" representation is chosen.

A successful **lockFocus** message must be balanced by a subsequent **unlockFocus** message to the same NSImage. These messages bracket the code that draws the image.

If **lockFocus** is unable to focus on the representation, it raises an NSImageCacheException.

See also: **– bestRepresentationForDevice:**, **– isValid**, **– prefersColorMatch**, **– representations**

lockFocusOnRepresentation:

– (void)**lockFocusOnRepresentation:**(NSImageRep *)*imageRepresentation*

Prepares for drawing of the *imageRepresentation* representation by making the off-screen window where it will be cached the current window and a coordinate system specific to the area where the image will be drawn the current coordinate system. If *imageRepresentation* is **nil**, **lockFocusOnRepresentation:** acts like **lockFocus**, setting focus to the best representation for the NSImage. Otherwise, *imageRepresentation* must be one of the representations in the NSImage.

A successful **lockFocusOnRepresentation:** message must be balanced by a subsequent **unlockFocus** message to the same NSImage. These messages bracket the code that draws the image.

If **lockFocusOnRepresentation:** is unable to focus on the representation, it raises an NSImageCacheException.

See also: **– isValid**

matchesOnMultipleResolution

– (BOOL)**matchesOnMultipleResolution**

Returns YES if the resolution of the device and the resolution specified for the image are considered to match if one is an integer multiple of the other, and NO if device and image resolutions are considered to match only if they are exactly the same. The default is YES.

name

– (NSString *)**name**

Returns the name assigned to the receiver, or **nil** if no name has been assigned.

prefersColorMatch

– (BOOL)**prefersColorMatch**

Returns YES if, when selecting the representation it will use, the NSImage first looks for one that matches the color capability of the rendering device (choosing a gray-scale representation for a monochrome device and a color representation for a color device), then if necessary narrows the selection by looking for one that matches the resolution of the device. If the return is NO, the NSImage first looks for a representation that matches the resolution of the device, then tries to match the representation to the color capability of the device. The default is YES.

recache

– (void)**recache**

Invalidates the off-screen caches of all representations and frees them. The next time any representation is composited, it will first be asked to redraw itself in the cache. NSCachedImageReps aren't destroyed by this method.

If an image is likely not to be used again, it's a good idea to free its caches, since that will reduce that amount of memory consumed by your program and therefore improve performance.

removeRepresentation:

– (void)**removeRepresentation:**(NSImageRep *)*imageRep*

Removes and releases the *imageRep* representation from the NSImage's list of representations.

See also: **– representations**

representations

– (NSArray *)**representations**

Returns an array containing all of the representations of the receiver.

scalesWhenResized

– (BOOL)**scalesWhenResized**

Returns YES if image representations are scaled to fit the size specified for the NSImage. If representations are not scalable, this method returns NO. The default is NO.

Representations created from data that specifies a size (for example, the "ImageLength" and "ImageWidth" fields of a TIFF representation or the bounding box of an EPS representation) will have the size the data specifies, which may differ from the size of the NSImage.

See also: **– setSize:**

setBackgroundColor:

– (void)**setBackgroundColor:**(NSColor *)*aColor*

Sets the background color of the image. The default is NSColor's **clearColor**, indicating a transparent background. The background color will be visible only for representations that don't completely cover all the pixels within the image when drawing. This method doesn't cause the receiver to recache itself.

See also: **– recache**

setCacheDepthMatchesImageDepth:

– (void)**setCacheDepthMatchesImageDepth:**(BOOL)*flag*

Sets whether the application's default depth limit applies to the off-screen windows where the NSImage's representations are cached. If *flag* is NO (the default), window depths are instead determined by the specifications of the representations. This method doesn't cause the receiver to recache itself.

See also: **– lockFocus**, **– recache**

setCachedSeparately:

– (void)**setCachedSeparately:**(BOOL)*flag*

Sets whether each image representation will be cached in its own off-screen window or in a window shared with other images. If *flag* is YES, each representation is guaranteed to be in a separate window. If *flag* is NO (the default), a representation can be cached together with other images, though in practice it might not be.

If an NSImage is to be resized frequently, it's more efficient to cache its representations separately.

This method doesn't invalidate any existing caches.

See also: **– recache**

setDataRetained:

– (void)**setDataRetained:**(BOOL)*flag*

Sets whether the NSImage retains the data needed to render the image. The default is NO. If the data is available in a file that won't be moved or deleted, or if responsibility for drawing the image is delegated to another object with a custom method, there's no reason for the NSImage to retain the data. However, if the NSImage reads image data from a file that could change, you may want to have it keep the data itself. Generally, this is useful to redraw the image to a device of different resolution.

If an image representation is created using **initByReferencingFile:**, the only data retained is the name of the source file.

setDelegate:

– (void)**setDelegate:**(id)*anObject*

Makes *anObject* the delegate of the receiver.

setFlipped:

– (void)**setFlipped:**(BOOL)*flag*

Determines whether the polarity of the y-axis is inverted when drawing an image. If *flag* is YES, the image will have its coordinate origin in the upper left corner and the positive y-axis will extend downward. This method affects only the coordinate system used to draw the image; it doesn't affect the coordinate system for specifying portions of the image for methods like **compositeToPoint:fromRect:operation:** or **dissolveToPoint:fromRect:fraction:**.This method doesn't cause the receiver to recache itself.

See also: **– recache**

setMatchesOnMultipleResolution:

– (void)**setMatchesOnMultipleResolution:**(BOOL)*flag*

Sets whether image representations with resolutions that are integral multiples of the resolution of the device are considered to match the device. The default is YES.

setName:

– (BOOL)**setName:**(NSString *)*aString*

Registers the receiver under the name specified by *aString*, provided that no other NSImage is registered using that name. If the receiver is already registered under another name, **setName:** first unregisters the prior name. **setName:** returns YES unless another NSImage is registered using the name specified by *aString*, in which case **setName:** simply returns NO.

See also: **+ imageNamed:**

setPrefersColorMatch:

– (void)**setPrefersColorMatch:**(BOOL)*flag*

Sets whether color matches are preferred over resolution matches when determining which representation to use. If *flag* is YES, the NSImage first tries to match the representation to the color capability of the rendering device (choosing a color representation for a color device and a gray-scale representation for a monochrome device), and then if necessary narrows the selection by trying to match the resolution of the representation

to the resolution of the device. If *flag* is NO, the NsImage first tries to match the representation to the resolution of the device, and then tries to match it to the color capability of the device. The default is YES.

setScalesWhenResized:

– (void)**setScalesWhenResized:**(BOOL)*flag*

Sets whether representations with sizes that differ from the size of the NSImage will be scaled to fit. If flag is YES, representations are scaled to fit. The default is NO.

Generally, representations that are created through NSImage methods (such as **initByReferencingFile:**) have the same size as the NSImage. However, a representation that's added with either **addRepresentation:** or **addRepresentations:** may have a different size, and representations created from data that specifies a size (for example, the "ImageLength" and "ImageWidth" fields of a TIFF representation or the bounding box of an EPS representation) will have the size specified.

This method doesn't cause the receiving NSImage to recache itself when it is next composited.

See also: **– setSize:**

setSize:

– (void)**setSize:**(NSSize)*aSize*

Sets the width and height of the image. The size referred to by *aSize* should be in units of the base coordinate system.

The size of an NSImage must be set before it can be used. You can change the size of an NSImage after it has been used, but changing it invalidates all its caches and frees them. When the image is next composited, the selected representation will draw itself in an off-screen window to recreate the cache.

If the size of the image hasn't already been set when the NSImage is provided with a representation, the size will be set from the data. The bounding box is used to determine the size of an NSEPSImageRep. The TIFF fields "ImageLength" and "ImageWidth" are used to determine the size of an NSBitmapImageRep.

See also: **– initWithSize:**, **– setScalesWhenResized:**

setUsesEPSOnResolutionMismatch:

– (void)**setUsesEPSOnResolutionMismatch:**(BOOL)*flag*

Sets whether EPS representations are preferred when there are no representations that match the resolution of the device. The default is NO.

See also: **– setMatchesOnMultipleResolution:**

size

– (NSSize)**size**

Returns the size of the image. If no size has been set, and no size can be determined from any of the NSImage's representations, the returned NSSize will have a width and height of 0.0.

TIFFRepresentation

– (NSData *)**TIFFRepresentation**

Returns a data object containing TIFF for all representations, using their default compressions.

TIFFRepresentationUsingCompression:factor:

– (NSData *)**TIFFRepresentationUsingCompression:**(NSTIFFCompression)*comp*
factor:(float)*aFloat*

Returns a data object containing TIFF for all representations, using the specified compression type and compression factor. Legal values for *comp* can be found in **NSBitmapImageRep.h**, and are described in "Tiff Compression" in NSBitmapImageRep's class description. *aFloat* provides a hint for those compression types that implement variable compression ratios; currently only JPEG compression uses a compression factor.

If the specified compression isn't applicable, no compression is used. If a problem is encountered during generation of the TIFF, **TIFFRepresentationUsingCompression:factor:** raises an exception.

See also: – **TIFFRepresentationUsingCompression:factor:** (NSBitmapImageRep)

unlockFocus

– (void)**unlockFocus**

Balances a previous **lockFocus** or **lockFocusOnRepresentation:** message. All successful **lockFocus** and **lockFocusOnRepresentation:** messages (those that don't raise an NSImageCacheException) must be followed by a subsequent **unlockFocus** message. Those that raise should never be followed by **unlockFocus**.

usesEPSOnResolutionMismatch

– (BOOL)**usesEPSOnResolutionMismatch**

Returns whether EPS representations are preferred when there are no representations that match the resolution of the device. The default is NO.

See also: **– matchesOnMultipleResolution**

Methods Implemented By the Delegate

imageDidNotDraw:inRect:

– (NSImage *)**imageDidNotDraw:**(id)*sender*
inRect:(NSRect)*aRect*

Implemented by the delegate to respond to a message sent by the *sender* (an NSImage) when the *sender* was unable, for whatever reason, to composite or lock focus on its image. The delegate can:

- return another NSImage to draw in the *sender*'s place,
- draw the image itself and return **nil**, or
- simply return **nil** to indicate that *sender* should give up the attempt at drawing the image.

NSImageCell

Inherits From: NSCell : NSObject

Conforms To: NSCopying, NSCoding (from NSCell)
NSObject (from NSObject)

Declared In: AppKit/NSImageCell.h

Class Description

An NSImageCell displays a single NSImage in a frame. This class provides methods for choosing the frame, and for aligning and scaling the image to fit the frame.

The object value of an NSImageCell must be an NSImage, so if you use NSCell's **setObjectValue:** method, be sure to supply an NSImage as an argument. Since an NSImage doesn't need to be converted for display, you won't use the NSCell methods relating to formatters.

An NSImageCell is usually associated with some kind of NSControl—an NSImageView, an NSMatrix, or an NSTableView. For more information, see the specifications for those classes.

Method Types

Aligning and scaling the image

– imageAlignment
– setImageAlignment:
– imageScaling
– setImageScaling:

Choosing the frame

– imageFrameStyle
– setImageFrameStyle:

Instance Methods

imageAlignment

– (NSImageAlignment)**imageAlignment**

Returns the position of the cell's image in the frame. For a list of possible alignments, see **setImageAlignment:**.

imageFrameStyle

– (NSImageFrameStyle)**imageFrameStyle**

Returns the style of frame that appears around the image. For a list of frame styles, see **setImageFrameStyle:**.

imageScaling

– (NSImageScaling)**imageScaling**

Returns the way that the cell's image alters to fit the frame. For a list of possible values, see **setImageScaling:**.

setImageAlignment:

– (void)**setImageAlignment:**(NSImageAlignment)*alignment*

Lets you specify the position of the image in the frame. The possible alignments are:

- NSImageAlignLeft
- NSImageAlignRight
- NSImageAlignCenter
- NSImageAlignTop
- NSImageAlignBottom
- NSImageAlignTopLeft
- NSImageAlignTopRight
- NSImageAlignBottomLeft
- NSImageAlignBottomRight

The default *alignment* is NSImageAlignCenter.

See also: – imageAlignment

setImageFrameStyle:

– (void)**setImageFrameStyle:**(NSImageFrameStyle)*frameStyle*

Lets you specify the kind of frame that borders the image. The possible styles are:

- NSImageFrameNone—an invisible frame
- NSImageFramePhoto—a thin black outline and a dropped shadow
- NSImageFrameGrayBezel—a gray, concave bezel that makes the image look sunken
- NSImageGroove—a thin groove that looks etched around the image
- NSImageFrameButton—a convex bezel that makes the image stand out in relief, like a button

The default *frameStyle* is NSImageFrameNone.

See also: – imageFrameStyle

setImageScaling:

– (void)**setImageScaling:**(NSImageScaling)*scaling*

Lets you specify the way that the image alters to fit the frame. The possible values are:

- NSScaleProportionally. If the image is too large, it shrinks to fit inside the frame. If the image is too small, it expands. The proportions of the image are preserved.
- NSScaleToFit. The image shrinks or expands, and its proportions distort, until it exactly fits the frame.
- NSScaleNone. The size and proportions of the image don't change. If the frame is too small to display the whole image, the edges of the image are trimmed off.

The default *scaling* is NSScaleProportionally.

The default *scaling* is NSScaleProportionally.

See also: – imageScaling

NSImageRep

Inherits From: NSObject

Conforms To: NSCoding
NSCopying
NSObject (NSObject)

Declared In: AppKit/NSImageRep.h

Class Description

NSImageRep is a semi-abstract superclass ("semi," because it has some instance variables and implementation of its own); each of its subclasses knows how to draw an image from a particular kind of source data. While an NSImageRep subclass can be used directly, it's typically used through an NSImage object. An NSImage manages a group of representations, choosing the best one for the current output device.

There are four subclasses defined in the Application Kit:

Subclass	Source Data
NSBitmapImageRep	Tag Image File Format (TIFF), Windows bitmap (BMP) and other bitmap data
NSCachedImageRep	A rendered image, usually in an off-screen window
NSCustomImageRep	A delegated method that can draw the image
NSEPSImageRep	Encapsulated PostScript code (EPS)

You can define other NSImageRep subclasses for objects that render images from other types of source information. New subclasses must be added to the NSImageRep class registry by invoking the **registerImageRepClass:** class method. The NSImageRep subclass informs the registry of the data types it can support through its **imageUnfilteredFileTypes**, **imageUnfilteredPasteboardTypes**, and **canInitWithData:** class methods. Once an NSImageRep subclass is registered, an instance of that subclass is created any time NSImage encounters the type of data handled by that subclass.

Subclasses which deal with file and pasteboard types should implement **imageUnfilteredFileTypes**, **imageUnfilteredPasteboardTypes**, **initWithData:**, **canInitWithData:**, and, if they have the ability to

read multiple images from a file, **imageRepsWithData:**. These last three should not do any filtering; all filtering is automatic.

Adopted Protocols

NSCoding

– encodeWithCoder:
– initWithCoder:

NSCopying

– copyWithZone:

Method Types

Creating an NSImageRep

+ imageRepsWithContentsOfFile:
+ imageRepsWithPasteboard:
+ imageRepWithContentsOfFile:
+ imageRepWithPasteboard:

Checking data types

+ canInitWithData:
+ canInitWithPasteboard:
+ imageFileTypes
+ imagePasteboardTypes
+ imageUnfilteredFileTypes
+ imageUnfilteredPasteboardTypes

Setting the size of the image

– setSize:
– size

Specifying information about the representation

– bitsPerSample
– colorSpaceName
– hasAlpha
– isOpaque
– pixelsHigh
– pixelsWide
– setAlpha:
– setBitsPerSample:
– setColorSpaceName:
– setOpaque:
– setPixelsHigh:
– setPixelsWide:

Drawing the image

– draw
– drawAtPoint:
– drawInRect:

Managing NSImageRep subclasses

+ imageRepClassForData:
+ imageRepClassForFileType:
+ imageRepClassForPasteboardType:
+ registeredImageRepClasses
+ registerImageRepClass:
+ unregisterImageRepClass:

Class Methods

canInitWithData:

+ (BOOL)**canInitWithData:**(NSData *)*data*

Overridden in subclasses to return YES if the receiver can initialize itself from *data*, and NO if it cannot. Note that this method doesn't need to do a comprehensive check; it should return NO only if it knows that the receiver can't initialize itself from *data*.

canInitWithPasteboard:

+ (BOOL)**canInitWithPasteboard:**(NSPasteboard *)*pasteboard*

Returns YES if the NSImageRep can handle the data represented by pasteboard, otherwise returns NO.

This method invokes the **imageUnfilteredPasteboardTypes** class method and checks the list of types returned by that method against the data types in pasteboard. If it finds a match, it returns YES. When creating a subclass of NSImageRep that accepts image data from a non-default pasteboard type, override the **imageUnfilteredPasteboardTypes** method to assure that this method returns the correct response.

imageFileTypes

+ (NSArray *)**imageFileTypes**

Returns an array of NSStrings representing all file types supported by NSImageRep or one of its subclasses. The list includes both those types returned by **the imageUnfilteredFileTypes** class method and those that can be converted to a supported type by a user-installed filter service. Don't override this method when subclassing NSImageRep—it always returns a valid list for any subclass of NSImageRep that correctly overrides the **imageUnfilteredFileTypes** method.

imagePasteboardTypes

+ (NSArray *)**imagePasteboardTypes**

Returns an array of NSStrings representing all pasteboard types supported by NSImageRep or one of its subclasses. The list includes both those types returned by the **imageUnfilteredPasteboardTypes** class method and those that can be converted by a user-installed filter service to a supported type. Don't override this method when subclassing NSImageRep—it always returns a valid list for any subclass of NSImageRep that correctly overrides the **imageUnfilteredPasteboardTypes** method.

imageRepClassForData:

+ (Class)**imageRepClassForData:**(NSData *)*data*

Returns the NSImageRep subclass that handles data of type *data*, or **Nil** if the NSImage class registry contains no subclasses that handle data of the specified type.

imageRepClassForFileType:

+ (Class)**imageRepClassForFileType:**(NSString *)*type*

Returns the NSImageRep subclass that handles files of type *type*, or **Nil** if the NSImage class registry contains no subclasses that handle files of the specified type.

imageRepClassForPasteboardType:

+ (Class)**imageRepClassForPasteboardType:**(NSString *)*type*

Returns the NSImageRep subclass that handles pasteboard data of type *type*, or **Nil** if the NSImage class registry contains no subclasses that handle pasteboard data of the specified type.

imageRepWithContentsOfFile:

+ (id)**imageRepWithContentsOfFile:**(NSString *)*filename*

If sent to the NSImageRep class object, this method returns a newly-allocated instance of a subclass of NSImageRep (chosen through the use of **imageRepClassForFileType:**) that's initialized with the contents of the file *filename*. If sent to a subclass of NSImageRep that recognizes the type of file specified by *filename*, it returns an instance of that subclass initialized with the contents of the file *filename*

imageRepWithContentsOfFile: returns **nil** in any of the following cases:

- The message is sent to the NSImageRep class object, and there are no subclasses in the NSImageRep class registry that handle data of the type indicated by *filename*.
- The message is sent to a subclass of NSImageRep, and that subclass doesn't handle data of the type indicated by *filename*.
- The NSImageRep subclass is unable to initialize itself with the contents of *filename*.

filename may be a full or relative pathname, and should include an extension that identifies the data type in the file. By default, the files handled are those with the extensions "tiff", "tif", "bmp", and "eps".

The NSImageRep subclass is initialized by creating an NSData object based on the contents of the file, then passing it to **imageRepWithData:**.

See also: **+ imageFileTypes**

imageRepWithPasteboard:

+ (id)**imageRepWithPasteboard:**(NSPasteboard *)*pasteboard*

If sent to the NSImageRep class object, this method returns a newly-allocated instance of a subclass of NSImageRep that's initialized with the data in *pasteboard*. If sent to a subclass of NSImageRep that recognizes the type of data contained in *pasteboard*, it returns an instance of that subclass initialized with the data in *pasteboard*

imageRepWithPasteboard: returns **nil** in any of the following cases:

- The message is sent to the NSImageRep class object, and there are no subclasses in the NSImageRep class registry that handle data of the type contained in *pasteboard*.

- The message is sent to a subclass of NSImageRep, and that subclass doesn't handle data of the type contained in *pasteboard*.
- The NSImageRep subclass is unable to initialize itself with the contents of *pasteboard*.

The NSImageRep subclass is initialized by creating an NSData object based on the data in pasteboard, then passing it to **imageRepWithData:**.

See also: **+ imagePasteboardTypes**

imageRepsWithContentsOfFile:

+ (NSArray *)**imageRepsWithContentsOfFile:**(NSString *)*filename*

If sent to the NSImageRep class object, this method returns an array of objects (all newly-allocated instances of a subclass of NSImageRep, chosen through the use of **imageRepClassForFileType:**) that have been initialized with the contents of the file *filename*. If sent to a subclass of NSImageRep that recognizes the type of file specified by *filename*, it returns an array of objects (all instances of that subclass) that have been initialized with the contents of the file *filename*

imageRepsWithContentsOfFile: returns **nil** in any of the following cases:

- The message is sent to the NSImageRep class object, and there are no subclasses in the NSImageRep class registry that handle data of the type indicated by *filename*.
- The message is sent to a subclass of NSImageRep, and that subclass doesn't handle data of the type indicated by *filename*.
- The NSImageRep subclass is unable to initialize itself with the contents of *filename*.

filename may be a full or relative pathname, and should include an extension that identifies the data type in the file. By default, the files handled are those with the extensions "tiff", "tif", "bmp", and "eps".

The NSImageRep subclass is initialized by creating an NSData object based on the contents of the file, then passing it to **imageRepsWithData:**.

See also: **+ imageFileTypes**

imageRepsWithPasteboard:

+ (NSArray *)**imageRepsWithPasteboard:**(NSPasteboard *)*pasteboard*

If sent to the NSImageRep class object, this method returns an array of objects (all newly-allocated instances of a subclass of NSImageRep) that have been initialized with the data in *pasteboard*. If sent to a subclass of NSImageRep that recognizes the type of data contained in *pasteboard*, it returns an array of objects (all instances of that subclass) initialized with the data in *pasteboard*

imageRepsWithPasteboard: returns **nil** in any of the following cases:

- The message is sent to the NSImageRep class object, and there are no subclasses in the NSImageRep class registry that handle data of the type contained in *pasteboard*.
- The message is sent to a subclass of NSImageRep, and that subclass doesn't handle data of the type contained in *pasteboard*.
- The NSImageRep subclass is unable to initialize itself with the contents of *pasteboard*.

The NSImageRep subclass is initialized by creating an NSData object based on the data in *pasteboard*, then passing it to **imageRepsWithData:**.

See also: + **imagePasteboardTypes**

imageUnfilteredFileTypes

+ (NSArray *)**imageUnfilteredFileTypes**

Returns an array of NSStrings representing all file types (extensions) supported by the NSImageRep. By default, the returned array is empty.

When creating a subclass of NSImageRep, override this method to return a list of strings representing the supported file types. For example, NSBitmapImageRep implements the following code for this method:

```
+ (NSArray *)imageUnfilteredFileTypes {
    static NSArray *types = nil;

    if (!types) types = [[NSArray alloc]
        initWithObjects:@"tiff", @"tif", @"bmp", nil];
    return types;
}
```

If your subclass supports the types supported by its superclass, you must explicitly get the array of types from the superclass and put them in the array returned by this method.

See also: + **imageFileTypes**, + **imageUnfilteredFileTypes** (NSImage)

imageUnfilteredPasteboardTypes

+ (NSArray *)**imageUnfilteredPasteboardTypes**

Returns an array representing all pasteboard types supported by the NSImageRep. By default, the returned array is empty.

When creating a subclass of NSImageRep, override this method to return a list representing the supported pasteboard types. For example, NSBitmapImageRep implements the following code for this method:

```
+ (NSArray *)imageUnfilteredPasteboardTypes {
    static NSArray *types = nil;
```

```
    if (!types) types = [[NSArray alloc] initWithObjects:NSTIFFPboardType, nil];
    return types;
}
```

If your subclass supports the types supported by its superclass, you must explicitly get the list of types from the superclass and add them to the array returned by this method.

See also: + **imagePasteboardTypes**, + **imageUnfilteredPasteboardTypes** (NSImage)

registerImageRepClass:

+ (void)**registerImageRepClass:**(Class)*imageRepClass*

Adds *imageRepClass* to the registry of available NSImageRep classes. This method posts the NSImageRepRegistryChangedNotification notification, along with the receiving object, to the default notification center.

A good place to add image representation classes to the registry is in the **load** class method.

See also: + **load** (NSObject)

registeredImageRepClasses

+ (NSArray *)**registeredImageRepClasses**

Returns an array containing the registered NSImageRep classes.

unregisterImageRepClass:

+ (void)**unregisterImageRepClass:**(Class)*imageRepClass*

Removes *imageRepClass* from the registry of available NSImageRep classes. This method posts the NSImageRepRegistryChangedNotification notification, along with the receiving object, to the default notification center.

Instance Methods

bitsPerSample

– (int)**bitsPerSample**

Returns the number of bits used to specify a single pixel in each component of the data.

colorSpaceName

– (NSString *)**colorSpaceName**

Returns the name if the image's color space, or NSCalibratedRGBColorSpace if no name has been assigned.

draw

– (BOOL)**draw**

Implemented by subclasses to draw the image at location (0.0, 0.0) in the current coordinate system. Subclass methods return YES if the image is successfully drawn, and NO if it isn't. This version of the method simply returns YES.

drawAtPoint:

– (BOOL)**drawAtPoint:**(NSPoint)*aPoint*

Sets the current coordinates to those indicated by *aPoint*, invokes the receiver's **draw** method draw the image at that point, then restores the current coordinates to their original setting. If *aPoint* is (0.0, 0.0), **drawAtPoint:** simply invokes **draw**.

This method returns NO without translating, scaling, or drawing if the size of the image has not been set. Otherwise it returns the value returned by the **draw** method, which indicates whether the image is successfully drawn.

See also: **– setSize:**

drawInRect:

– (BOOL)**drawInRect:**(NSRect)*rect*

Draws the image so that it fits inside the rectangle referred to by *rect*. The current coordinates are set to the point specified in the rectangle and are scaled so the image will fit within the rectangle. The receiver's **draw** method is then invoked to draw the image. After draw has been invoked, the current coordinates and scale factors are restored to their original settings.

This method returns NO without translating, scaling, or drawing if the size of the image has not been set. Otherwise it returns the value returned by the **draw** method, which indicates whether the image is successfully drawn.

See also: **– setSize:**

hasAlpha

– (BOOL)**hasAlpha**

Returns YES if the receiver has been informed that the image has a coverage component (alpha), and NO if not.

isOpaque

– (BOOL)**isOpaque**

Returns YES if the receiver is opaque; NO otherwise. Use this method to test whether an NSImageRep completely covers the area within the rectangle returned by **size:**. Use the method **setOpaque:** to set the value returned by this method.

pixelsHigh

– (int)**pixelsHigh**

Returns the height of the image in pixels, as specified in the image data.

See also: **– size**

pixelsWide

– (int)**pixelsWide**

Returns the width of the image in pixels, as specified in the image data.

See also: **– size**

setAlpha:

– (void)**setAlpha:**(BOOL)*flag*

Informs the NSImageRep whether the image has an alpha component. *flag* should be YES if it does, and NO if it doesn't.

setBitsPerSample:

– (void)**setBitsPerSample:**(int)*anInt*

Informs the NSImageRep that the image has *anInt* bits of data for each pixel in each component.

setColorSpaceName:

– (void)**setColorSpaceName:**(NSString *)*string*

Informs the receiver of the image's color space. By default, an NSImageRep's color space name is NSCalibratedRGBColorSpace. Color space names are defined as part of the NSColor class, in NSGraphics.h. The following are valid color space names:

NSCalibratedWhiteColorSpace
NSCalibratedBlackColorSpace
NSCalibratedRGBColorSpace
NSDeviceWhiteColorSpace
NSDeviceBlackColorSpace
NSDeviceRGBColorSpace
NSDeviceCMYKColorSpace
NSNamedColorSpace
NSCustomColorSpace

setOpaque:

– (void)**setOpaque:**(BOOL)*flag*

Sets opacity of the NSImageRep's image. If flag is YES, the image is opaque.

setPixelsHigh:

– (void)**setPixelsHigh:**(int)*anInt*

Informs the NSImageRep that the data specifies an image *anInt* pixels high.

See also: **– setSize:**

setPixelsWide:

– (void)**setPixelsWide:**(int)*anInt*

Informs the NSImageRep that the data specifies an image *anInt* pixels wide.

See also: **– setSize:**

setSize:

– (void)**setSize:**(NSSize)*aSize*

Sets the size of the image in units of the base coordinate system. This determines the size of the image when it's rendered; it's not necessarily the same as the width and height of the image in pixels as specified in the image data. You must set the image size before you can render it.

See also: **– draw**, **– setPixelsHigh:**, **– setPixelsWide:**

size

– (NSSize)**size**

Returns the size of the image in units of the base coordinate system. This is the size of the image when it's rendered; it's not necessarily the same as the width and height of the image in pixels as specified in the image data.

See also: **– pixelsHigh**, **– pixelsWide**

Notifications

NSImageRepRegistryDidChangeNotification

Posted whenever the NSImageRep class registry changes.

This notification contains a notification object but no userInfo dictionary. The notification object is the image class that is registered or unregistered.

NSImageView

Inherits From: NSControl: NSView: NSResponder: NSObject

Conforms To: NSCoding (from NSResponder)
NSObject (from NSObject)

Declared In: AppKit/NSImageView.h

Class Description

An NSImageView displays a single NSImage in a frame. The NSImageView class provides methods for choosing the image, choosing the frame, and for aligning and scaling the image to fit the frame.

For an NSControl, NSImageView is quite limited in its ability to respond to user events: the only thing a user can do is drag in a new image. When it receives the new image, the NSImageView replaces its old image and sends its action message to its target. Even this low level of interactivity can be disabled: you can send the NSImageView the message **setEditable:**NO.

For more information, see the class specification for NSImageCell.

Method Types

Choosing the image

– image
– setImage:

Choosing the frame

– imageFrameStyle
– setImageFrameStyle:

Aligning and scaling the image

– imageAlignment
– setImageAlignment:
– imageScaling
– setImageScaling:

Responding to user events

– isEditable
– setEditable:

Instance Methods

image

– (NSImage *)**image**

Returns the NSImage displayed by the NSImageView.

See also: – **setImage:**

imageAlignment

– (NSImageAlignment)**imageAlignment**

Returns the position of the cell's image in the frame. For a list of possible alignments, see **setImageAlignment:**.

imageFrameStyle

– (NSImageFrameStyle)**imageFrameStyle**

Returns the style of frame that appears around the image. For a list of frame styles, see **setImageFrameStyle:**.

imageScaling

– (NSImageScaling)**imageScaling**

Returns the way that the cell's image alters to fit the frame. For a list of possible values, see **setImageScaling:**.

isEditable

– (BOOL)**isEditable**

Returns whether the user can drag a new image into the frame. The default is YES.

See also: – **setEditable**:

setEditable:

– (void)**setEditable:**(BOOL)*flag*

Specifies whether the user can drag a new image into the frame.

See also: – **isEditable**

setImage:

– (void)**setImage:**(NSImage *)*image*

Lets you specify the image that the NSImageView displays.

See also: **– image**

setImageAlignment:

– (void)**setImageAlignment:**(NSImageAlignment)*alignment*

Lets you specify the position of the image in the frame. The possible alignments are:

- NSImageAlignLeft
- NSImageAlignRight
- NSImageAlignCenter
- NSImageAlignTop
- NSImageAlignBottom
- NSImageAlignTopLeft
- NSImageAlignTopRight
- NSImageAlignBottomLeft
- NSImageAlignBottomRight

The default *alignment* is NSImageAlignCenter.

See also: – imageAlignment

setImageFrameStyle:

– (void)**setImageFrameStyle:**(NSImageFrameStyle)*frameStyle*

Lets you specify the kind of frame that borders the image . The possible styles are:

- NSImageFrameNone—an invisible frame
- NSImageFramePhoto—a thin black outline and a dropped shadow
- NSImageFrameGrayBezel—a gray, concave bezel that makes the image look sunken
- NSImageGroove—a thin groove that looks etched around the image
- NSImageFrameButton—a convex bezel that makes the image stand out in relief, like a button

The default *frameStyle* is NSImageFrameNone.

See also: – imageFrameStyle

setImageScaling:

– (void)**setImageScaling:**(NSImageScaling)*scaling*

Lets you specify the way that the image alters to fit the frame. The possible values are:

- NSScaleProportionally. If the image is too large, it shrinks to fit inside the frame. If the image is too small, it expands. The proportions of the image are preserved.
- NSScaleToFit. The image shrinks or expands, and its proportions distort, until it exactly fits the frame.
- NSScaleNone. The size and proportions of the image don't change. If the frame is too small to display the whole image, the edges of the image are trimmed off.

The default *scaling* is NSScaleProportionally.

See also: – imageScaling

NSInputManager

Inherits From: NSObject

Conforms To: NSTextInput
NSObject (NSObject)

Declared In: AppKit/NSInputManager.h

Class Description

Most programs never need to interact with an input manager. The system text object, and all UI objects that accept textual input, already deal with this typically through NSResponder.

Adopted Protocols

<<Forthcoming>>

Method Types

<<Forthcoming>>

Class Methods

currentInputManager

+ (NSInputManager *)**currentInputManager**

<<forthcoming>>

Instance Methods

initWithName:host:

– (NSInputManager *)**initWithName:**(NSString *)*inputServerName*
host:(NSString *)*hostName*

<<forthcoming>>

localizedInputManagerName

– (NSString *)**localizedInputManagerName**

<<forthcoming>>

markedTextSelectionChanged:sender:

– (void)**markedTextSelectionChanged:**(NSRange)*aRange*
sender:(id)*sender*

<<forthcoming>>

markedTextWillBeAbandoned:

– (void)**markedTextWillBeAbandoned:**(id)*sender*

<<forthcoming>>

wantsToInterpretAllKeystrokes

– (BOOL)**wantsToInterpretAllKeystrokes**

<<forthcoming>>

NSInputServer

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: AppKit/NSInputServer.h

Class Description

<<forthcoming>>

Adopted Protocols

<<Forthcoming>>

Method Types

<<Forthcoming>>

Instance Methods

activeConversationWillChange:oldConversation:newConversation:

– (void)**activeConversationWillChange:**(id)*sender*
oldConversation:(long)*conv*
newConversation:(long)*new*

<<forthcoming>>

canBeDisabled

– (BOOL)**canBeDisabled**

<<forthcoming>>

cancelInput:conversation:

– (void)**cancelInput:**(id)*sender* **conversation:**(long)*conv*

<<forthcoming>>

doCommandBySelector:sender:conversation:

– (void)**doCommandBySelector:**(SEL)*aSelector*
sender:(id)*sender*
conversation:(long)*conv*

<<forthcoming>>

initWithDelegate:name:

– **initWithDelegate:**(id)*aDelegate* **name:**(NSString *)*name*

<<forthcoming>>

insertText:sender:conversation:

– (void)**insertText:**(NSString *)*aString*
sender:(id)*sender*
conversation:(long)*conv*

<<forthcoming>>

markedTextSelectionChanged:sender:conversation:

– (void)**markedTextSelectionChanged:**(NSRange)*aRange*
sender:(id)*sender*
conversation:(long)*conv*

<<forthcoming>>

markedTextWillBeAbandoned:conversation:

– (void)**markedTextWillBeAbandoned:**(id)*sender* **conversation:**(long)*conv*

<<forthcoming>>

senderDidBecomeActive:

– *(void)***senderDidBecomeActive:***(id)sender*

<<forthcoming>>

senderDidResignActive:

– *(void)***senderDidResignActive:***(id)sender*

<<forthcoming>>

setActivated:sender:

– (void)**setActivated:**(BOOL)*flag* **sender:**(id)*sender*

<<forthcoming>>

terminate:

– (void)**terminate:**(id)*sender*

<<forthcoming>>

wantsToInterpretAllKeystrokes

– (BOOL)**wantsToInterpretAllKeystrokes**

<<forthcoming>>

NSLayoutManager

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: AppKit/NSLayoutManager.h

Class Description

An NSLayoutManager coordinates the layout and display of characters held in an NSTextStorage object. It maps Unicode character codes to glyphs, sets the glyphs in a series of NSTextContainers, and displays them in a series of NSTextViews. In addition to its core function of laying out text, an NSLayoutManager coordinates its NSTextViews, provides services to those NSTextViews to support NSRulerViews for editing paragraph styles, and handles the layout and display of text attributes not inherent in glyphs (such as underline or strikethrough). You can create a subclass of NSLayoutManager to handle additional text attributes, whether inherent or not.

Method Types

Creating an instance

– init

Setting the text storage

– setTextStorage:
– textStorage
– replaceTextStorage:

Setting text containers

– textContainers
– addTextContainer:
– insertTextContainer:atIndex:
– removeTextContainerAtIndex:

Invalidating glyphs and layout

– invalidateGlyphsForCharacterRange:changeInLength: actualCharacterRange:
– invalidateLayoutForCharacterRange:isSoft:actualCharacterRange:
– invalidateDisplayForCharacterRange:
– invalidateDisplayForGlyphRange:
– textContainerChangedGeometry:
– textStorage:edited:range:changeInLength:invalidatedRange:

Turning background layout on/off

– setBackgroundLayoutEnabled:
– backgroundLayoutEnabled

Accessing glyphs

– insertGlyph:atGlyphIndex:characterIndex:
– glyphAtIndex:
– glyphAtIndex:isValidIndex:
– replaceGlyphAtIndex:withGlyph:
– getGlyphs:range:
– deleteGlyphsInRange:
– numberOfGlyphs

Mapping characters to glyphs

– setCharacterIndex:forGlyphAtIndex:
– characterIndexForGlyphAtIndex:
– characterRangeForGlyphRange:actualGlyphRange:
– glyphRangeForCharacterRange:actualCharacterRange:

Setting glyph attributes

– setIntAttribute:value:forGlyphAtIndex:
– intAttribute:forGlyphAtIndex:

Handling layout for text containers

– setTextContainer:forGlyphRange:
– glyphRangeForTextContainer:
– textContainerForGlyphAtIndex:effectiveRange:
– usedRectForTextContainer:

Handling line fragment rectangles

– setLineFragmentRect:forGlyphRange:usedRect:
– lineFragmentRectForGlyphAtIndex:effectiveRange:
– lineFragmentUsedRectForGlyphAtIndex:effectiveRange:
– setExtraLineFragmentRect:usedRect:textContainer:
– extraLineFragmentRect
– extraLineFragmentUsedRect
– extraLineFragmentTextContainer
– setDrawsOutsideLineFragment:forGlyphAtIndex:
– drawsOutsideLineFragmentForGlyphAtIndex:

Layout of glyphs

– setLocation:forStartOfGlyphRange:
– locationForGlyphAtIndex:
– rangeOfNominallySpacedGlyphsContainingIndex:
– rectArrayForCharacterRange:withinSelectedCharacterRange: inTextContainer:rectCount:
– rectArrayForGlyphRange:withinSelectedGlyphRange: inTextContainer:rectCount:
– boundingRectForGlyphRange:inTextContainer:
– glyphRangeForBoundingRect:inTextContainer:
– glyphRangeForBoundingRectWithoutAdditionalLayout: inTextContainer:
– glyphIndexForPoint:inTextContainer: fractionOfDistanceThroughGlyph:

Display of special glyphs

– setNotShownAttribute:forGlyphAtIndex:
– notShownAttributeForGlyphAtIndex:
– setShowsInvisibleCharacters:
– showsInvisibleCharacters
– setShowsControlCharacters:
– showsControlCharacters

Controlling hyphenation

– setHyphenationFactor:
– hyphenationFactor

Finding unlaid characters/glyphs

– getFirstUnlaidCharacterIndex:glyphIndex:

Using screen fonts

– setUsesScreenFonts:
– usesScreenFonts
– substituteFontForFont:

Handling rulers

– rulerAccessoryViewForTextView:paragraphStyle:ruler:enabled:
– rulerMarkersForTextView:paragraphStyle:ruler:

Managing the responder chain

– layoutManagerOwnsFirstResponderInWindow:
– firstTextView
– textViewForBeginningOfSelection

Drawing

– drawBackgroundForGlyphRange:atPoint:
– drawGlyphsForGlyphRange:atPoint:
– drawUnderlineForGlyphRange:underlineType:baselineOffset: lineFragmentRect:lineFragmentGlyphRange:containerOrigin:
– underlineGlyphRange:underlineType:lineFragmentRect: lineFragmentGlyphRange:containerOrigin:

Setting the delegate

– setDelegate:
– delegate

Instance Methods

addTextContainer:

– (void)**addTextContainer:**(NSTextContainer *)*aTextContainer*

Appends *aTextContainer* to the series of NSTextContainers where the receiver arranges text. Invalidates glyphs and layout as needed, but doesn't perform glyph generation or layout.

See also: – **insertTextContainer:atIndex:**, – **removeTextContainerAtIndex:**, – **textContainers**, – **invalidateGlyphsForCharacterRange:changeInLength:actualCharacterRange:**, – **invalidateLayoutForCharacterRange:isSoft:actualCharacterRange:**

backgroundLayoutEnabled

– (BOOL)**backgroundLayoutEnabled**

Returns YES if the receiver generates glyphs and lays out text when the application's run loop is idle, NO if it only performs glyph generation and layout when necessary.

See also: – **setBackgroundLayoutEnabled:**

boundingRectForGlyphRange:inTextContainer:

– (NSRect)**boundingRectForGlyphRange:**(NSRange)*glyphRange* **inTextContainer:** (NSTextContainer *)*aTextContainer*

Returns a single bounding rectangle enclosing all glyphs and other marks drawn in *aTextContainer* for *glyphRange*, including glyphs that draw outside their line fragment rectangles and text attributes such as underlining. This method is useful for determining the area that needs to be redrawn when a range of glyphs changes.

Performs glyph generation and layout if needed.

See also: – **glyphRangeForTextContainer:**, – **drawsOutsideLineFragmentForGlyphAtIndex:**

characterIndexForGlyphAtIndex:

– (unsigned int)**characterIndexForGlyphAtIndex:**(unsigned int)*glyphIndex*

Returns the index in the NSTextStorage for the first character mapped to the glyph at *glyphIndex* within the receiver. In many cases it's better to use the range-mapping methods, **characterRangeForGlyphRange: actualGlyphRange:** and **glyphRangeForCharacterRange:actualCharacterRange:**, which provide more comprehensive information.

Performs glyph generation if needed.

characterRangeForGlyphRange:actualGlyphRange:

– (NSRange)**characterRangeForGlyphRange:**(NSRange)*glyphRange* **actualGlyphRange:** (NSRange *)*actualGlyphRange*

Returns the range for the characters in the receiver's text store that are mapped to the glyphs in *glyphRange*. If *actualGlyphRange* is non-NULL, expands the requested range as needed so that it identifies all glyphs mapped to those characters and returns the new range by reference in *actualGlyphRange*.

Suppose the text store begins with the character "Ö" and the glyph cache contains "O" and "¨". If you get the character range for the glyph range {0, 1} or {1, 1}, *actualGlyphRange* is returned as {0, 2}, indicating that both of the glyphs are mapped to the character "Ö".

Performs glyph generation if needed.

See also: – **characterIndexForGlyphAtIndex:**, – **glyphRangeForCharacterRange: actualCharacterRange:**

delegate

– (id)**delegate**

Returns the receiver's delegate.

See also: – **setDelegate:**

deleteGlyphsInRange:

– (void)**deleteGlyphsInRange:**(NSRange)*glyphRange*

Deletes the glyphs in *glyphRange*.

This method is for use by the glyph generation mechanism, and doesn't perform any invalidation or generation of the glyphs or layout. You should never directly invoke this method.

See also: – **insertGlyph:atGlyphIndex:characterIndex:**

drawBackgroundForGlyphRange:atPoint:

– (void)**drawBackgroundForGlyphRange:**(NSRange)*glyphRange* **atPoint:** (NSPoint)*containerOrigin*

Draws background marks for *glyphRange*, which must lie completely within a single NSTextContainer. *containerOrigin* indicates the position of the NSTextContainer in the coordinate system of the NSView being drawn. This method must be invoked with the graphics focus locked on that NSView.

Background marks are such things as selection highlighting, text background color, and any background for marked text.

Performs glyph generation and layout if needed.

See also: – **drawGlyphsForGlyphRange:atPoint:**, – **glyphRangeForTextContainer:**, – **textContainerOrigin** (NSTextView)

drawGlyphsForGlyphRange:atPoint:

– (void)**drawGlyphsForGlyphRange:**(NSRange)*glyphRange* **atPoint:**(NSPoint)*containerOrigin*

Draws the glyphs in *glyphRange*, which must lie completely within a single NSTextContainer. *containerOrigin* indicates the position of the NSTextContainer in the coordinate system of the NSView being drawn. This method must be invoked with the graphics focus locked on that NSView.

Performs glyph generation and layout if needed.

See also: – **drawBackgroundForGlyphRange:atPoint:**, – **glyphRangeForTextContainer:**, – **textContainerOrigin** (NSTextView)

drawsOutsideLineFragmentForGlyphAtIndex:

– (BOOL)**drawsOutsideLineFragmentForGlyphAtIndex:**(unsigned int)*glyphIndex*

Returns YES if the glyph at *glyphIndex* exceeds the bounds of the line fragment where it's laid out, NO otherwise. This can happen when text is set at a fixed line height. For example, if the user specifies a fixed line height of 12 points and sets the font size to 24 points, the glyphs will exceed their layout rectangles.

Glyphs that draw outside their line fragment rectangles aren't considered when calculating enclosing rectangles with the **rectArrayForCharacterRange:withinSelectedCharacterRange:inTextContainer: rectCount:** and **rectArrayForGlyphRange:withinSelectedGlyphRange:inTextContainer:rectCount:** methods. They are, however, considered by **boundingRectForGlyphRange:inTextContainer:**

Performs glyph generation and layout if needed.

drawUnderlineForGlyphRange:underlineType:baselineOffset:lineFragmentRect: lineFragmentGlyphRange:containerOrigin:

– (void)**drawUnderlineForGlyphRange:**(NSRange)*glyphRange*
underlineType:(int)*underlineType*
baselineOffset:(float)*baselineOffset*
lineFragmentRect:(NSRect)*lineRect*
lineFragmentGlyphRange:(NSRange)*lineGlyphRange*
containerOrigin:(NSPoint)*containerOrigin*

Draws underlining for the glyphs in *glyphRange*, which must belong to a single line fragment rectangle (as returned by **lineFragmentRectForGlyphAtIndex:effectiveRange:**). *underlineType* indicates the style of underlining to draw; NSLayoutManager accepts only NSSingleUnderlineStyle, but subclasses can define their own underline styles. *baselineOffset* indicates how far below the text baseline the underline should be drawn; it's usually a positive value. *lineRect* is the line fragment rectangle containing the glyphs to draw underlining for, and *lineGlyphRange* is the range of all glyphs within that line fragment rectangle. *containerOrigin* is the origin of the line fragment rectangle's NSTextContainer in its NSTextView.

This method is invoked automatically by **underlineGlyphRange:...**; you should rarely need to invoke it directly.

See also: – **textContainerForGlyphAtIndex:effectiveRange:**, – **textContainerOrigin** (NSTextView)

extraLineFragmentRect

– (NSRect)**extraLineFragmentRect**

Returns the rectangle defining the extra line fragment for the insertion point at the end of a text (either in an empty text or after a final paragraph separator). The rectangle is defined in the coordinate system of its NSTextContainer. Returns NSZeroRect if there is no such rectangle.

See also: – **extraLineFragmentUsedRect**, – **extraLineFragmentTextContainer**, – **setExtraLineFragmentRect:usedRect:textContainer:**

extraLineFragmentTextContainer

– (NSTextContainer *)**extraLineFragmentTextContainer**

Returns the NSTextContainer that contains the extra line fragment rectangle, or **nil** if there is no extra line fragment rectangle. This rectangle is used to display the insertion point for the insertion point at the end of a text (either in an empty text or after a final paragraph separator).

See also: – **extraLineFragmentRect**, – **extraLineFragmentUsedRect**, – **setExtraLineFragmentRect:usedRect:textContainer:**

extraLineFragmentUsedRect

– (NSRect)**extraLineFragmentUsedRect**

Returns the rectangle enclosing the insertion point drawn in the extra line fragment rectangle. The rectangle is defined in the coordinate system of its NSTextContainer. Returns NSZeroRect if there is no extra line fragment rectangle.

The extra line fragment used rectangle is twice as wide (or tall) as the NSTextContainer's line fragment padding, with the insertion point itself in the middle.

See also: – **extraLineFragmentRect**, – **extraLineFragmentTextContainer**, – **setExtraLineFragmentRect:usedRect:textContainer:**

firstTextView

– (NSTextView *)**firstTextView**

Returns the first NSTextView in the receiver's series of text views. This is the object of various NSText and NSTextView notifications posted.

getFirstUnlaidCharacterIndex:glyphIndex:

– (void)**getFirstUnlaidCharacterIndex:**(unsigned int *)*charIndex* **glyphIndex:** (unsigned int *)*glyphIndex*

Returns by reference in *charIndex* and *glyphIndex* the indexes for the first character and glyph that have invalid layout information. Either parameter may be NULL, in which case the receiver simply ignores it.

getGlyphs:range:

– (unsigned int)**getGlyphs:**(NSGlyph *)*glyphArray* **range:**(NSRange)*glyphRange*

Fills *glyphArray* with displayable glyphs from *glyphRange* and returns the actual number of glyphs filled (which may be smaller than *glyphRange*'s length if some glyphs aren't drawn—for example, tab and newline characters). Raises an NSRangeException if the range specified exceeds the bounds of the actual glyph range for the receiver.

Performs glyph generation if needed.

See also: – **glyphAtIndex:**, – **glyphAtIndex:isValidIndex:**, – **notShownAttributeForGlyphAtIndex:**

glyphAtIndex:

– (NSGlyph)**glyphAtIndex:**(unsigned int)*glyphIndex*

Returns the glyph at *glyphIndex*. Raises an NSRangeException if *glyphIndex* is out of bounds.

Performs glyph generation if needed. To avoid an exception with **glyphAtIndex:** you must first check the glyph index against the number of glyphs, which requires generating *all* glyphs. Another method, **glyphAtIndex:isValidIndex:**, generates glyphs only up to the one requested, so using it can be more efficient.

See also: – **getGlyphs:range:**

glyphAtIndex:isValidIndex:

– (NSGlyph)**glyphAtIndex:**(unsigned int)*glyphIndex* **isValidIndex:**(BOOL *)*flag*

If *glyphIndex* is valid, returns the glyph at *glyphIndex* and sets *flag* to YES. Otherwise sets *flag* to NO (in which case the return value is meaningless).

Performs glyph generation if needed.

See also: – **getGlyphs:range:**, – **glyphAtIndex:**

glyphIndexForPoint:inTextContainer:fractionOfDistanceThroughGlyph:

– (unsigned int)**glyphIndexForPoint:**(NSPoint)*aPoint*
inTextContainer:(NSTextContainer *)*aTextContainer*
fractionOfDistanceThroughGlyph:(float *)*partialFraction*

Returns the index for the glyph nearest *aPoint* within *aTextContainer*. *aPoint* is expressed in *aTextContainer*'s coordinate system. If *partialFraction* is non-NULL the ratio of the distance into the glyph relative to the next glyph (in the appropriate sweep direction) is returned by reference in *partialFraction*.

Note: NSLayoutManager currently supports only left-to-right sweep.

For purposes such as dragging out a selection or placing the insertion point, a partial percentage less than or equal to 0.5 indicates that *aPoint* should be considered as falling before the glyph index returned; a partial percentage greater than 0.5 indicates that it should be considered as falling after the glyph index returned. If the nearest glyph doesn't lie under *aPoint* at all (for example, if *aPoint* is beyond the beginning or end of a line) this ratio will be 0 or 1.

Suppose the glyph stream contains the glyphs "A" and "b", with the width of "A" being 13 points. If the user clicks at a location 8 points into "A", *partialFraction* is 8 ÷ 13, or 0.615. In this case, the point given should be considered as falling between "A" and "b" for purposes such as dragging out a selection or placing the insertion point.

Performs glyph generation and layout if needed.

glyphRangeForBoundingRect:inTextContainer:

– (NSRange)**glyphRangeForBoundingRect:**(NSRect)*aRect* **inTextContainer:**
(NSTextContainer *)*aTextContainer*

Returns the smallest contiguous range for glyphs that are laid out wholly or partially within *aRect* in *aTextContainer*. The range returned can include glyphs that don't fall inside or intersect *aRect*, though the first and last glyphs in the range always do. This method is used to determine which glyphs need to be displayed within a given rectangle.

Performs glyph generation and layout if needed.

See also: – **glyphRangeForBoundingRectWithoutAdditionalLayout:inTextContainer:**

glyphRangeForBoundingRectWithoutAdditionalLayout:inTextContainer:

– (NSRange)**glyphRangeForBoundingRectWithoutAdditionalLayout:**(NSRect)*bounds*
inTextContainer:(NSTextContainer *)*container*

Returns the smallest contiguous range for glyphs that are laid out wholly or partially within *aRect* in *aTextContainer*. The range returned can include glyphs which don't fall inside or intersect *aRect*, though the first and last glyphs in the range always do.

Unlike **glyphRangeForBoundingRect:inTextContainer:**, this method doesn't perform glyph generation or layout. Its results, though faster, can be incorrect. This method is primarily for use by NSTextView; you should rarely need to use it yourself.

See also: – **glyphRangeForBoundingRect:inTextContainer:**

glyphRangeForCharacterRange:actualCharacterRange:

– (NSRange)**glyphRangeForCharacterRange:**(NSRange)*charRange* **actualCharacterRange:** (NSRange *)*actualCharRange*

Returns the range for the glyphs mapped to the characters of the text store in *charRange*. If *actualCharRange* is non-NULL, expands the requested range as needed so that it identifies all characters mapped to those glyphs and returns the new range by reference in *actualCharRange*.

Suppose the text store contains the characters "n˜" and the glyph cache contains "ñ". If you get the glyph range for the character range {0, 1} or {1, 1}, *actualCharRange* is returned as {0, 2}, indicating both of the characters mapped to the glyph "ñ".

Performs glyph generation if needed.

See also: – **characterIndexForGlyphAtIndex:**

glyphRangeForTextContainer:

– (NSRange)**glyphRangeForTextContainer:**(NSTextContainer *)*aTextContainer*

Returns the range for glyphs laid out within *aTextContainer*.

Performs glyph generation and layout if needed.

hyphenationFactor

– (float)**hyphenationFactor**

<forthcoming>

See also: – setHyphenationFactor:

init

– (id)**init**

Initializes the receiver, a newly created NSLayoutManager object. This is the designated initializer for the NSLayoutManager class. Returns **self**.

See also: – **addLayoutManager:** (NSTextStorage), – **addTextContainer:**

insertGlyph:atGlyphIndex:characterIndex:

– (void)**insertGlyph:**(NSGlyph)*aGlyph*
atGlyphIndex:(unsigned int)*glyphIndex*
characterIndex:(unsigned int)*charIndex*

Inserts *aGlyph* into the glyph cache at *glyphIndex* and maps it to the character at *charIndex*. If the glyph is mapped to several characters, *charIndex* should indicate the first character that it's mapped to.

This method is for use by the glyph generation mechanism, and doesn't perform any invalidation or generation of the glyphs or layout. You should never directly invoke this method.

See also: – **deleteGlyphsInRange:**, – **replaceGlyphAtIndex:withGlyph:**

insertTextContainer:atIndex:

– (void)**insertTextContainer:**(NSTextContainer *)*aTextContainer* **atIndex:**(unsigned int)*index*

Inserts *aTextContainer* into the series of text containers at *index*, and invalidates layout for all subsequent NSTextContainer's. Also invalidates glyph information as needed.

See also: – **addTextContainer:**, – **removeTextContainerAtIndex:**, – **textContainers**

intAttribute:forGlyphAtIndex:

– (int)**intAttribute:**(int)*attributeTag* **forGlyphAtIndex:**(unsigned int)*glyphIndex*

Returns the value of the attribute identified by *attributeTag* for the glyph at *glyphIndex*.

Subclasses that define their own custom attributes must override this method to access their own storage for the attribute values. Non-negative tags are reserved by Apple; you can define your own attributes with negative tags and set values using **setIntAttribute:value:forGlyphAtIndex:**.

invalidateDisplayForCharacterRange:

– (void)**invalidateDisplayForCharacterRange:**(NSRange)*charRange*

<forthcoming>

invalidateDisplayForGlyphRange:

– (void)**invalidateDisplayForGlyphRange:**(NSRange)*glyphRange*

Marks the glyphs in *glyphRange* as needing display, as well as the appropriate regions of the NSTextViews that display those glyphs (using NSView's **setNeedsDisplayInRect:**). You should rarely need to invoke this method.

invalidateGlyphsForCharacterRange:changeInLength:actualCharacterRange:

– (void)**invalidateGlyphsForCharacterRange:**(NSRange)*charRange*
changeInLength:(int)*lengthChange*
actualCharacterRange:(NSRange *)*actualCharRange*

Invalidates the cached glyphs for the characters in *charRange* and adjusts the remaining glyph-to-character mapping according to *lengthChange*, which indicates the number of characters added to or removed from the text store. If non-NULL, *actualCharRange* is set to the range of characters mapped to the glyphs just invalidated. This can be larger than the range of characters given due to the effect of context on glyphs and layout.

You should rarely need to invoke this method. It only invalidates glyph information, and performs no glyph generation or layout. Because invalidating glyphs also invalidates layout, after invoking this method you should also invoke **invalidateLayoutForCharacterRange:isSoft:actualCharacterRange:**, passing *charRange* as the first argument and NO as the flag to the **isSoft:** keyword.

invalidateLayoutForCharacterRange:isSoft:actualCharacterRange:

– (void)**invalidateLayoutForCharacterRange:**(NSRange)*charRange*
isSoft:(BOOL)*flag*
actualCharacterRange:(NSRange *)*actualCharRange*

Invalidates the layout information for the glyphs mapped to the characters in *charRange*. If *flag* is YES, attempts to save some layout information to avoid recalculation; if flag is NO, saves no layout information. You should typically pass NO for *flag*. If non-NULL, *actualCharRange* is set to the range of characters mapped to the glyphs whose layout information has been invalidated. This can be larger than the range of characters given due to the effect of context on glyphs and layout.

This method only invalidates information; it performs no glyph generation or layout. You should rarely need to invoke this method.

See also: – **invalidateGlyphsForCharacterRange:changeInLength:actualCharacterRange:**

layoutManagerOwnsFirstResponderInWindow:

– (BOOL)**layoutManagerOwnsFirstResponderInWindow:**(NSWindow *)*aWindow*

Returns YES if the first responder in *aWindow* is an NSTextView associated with the receiver, NO otherwise.

lineFragmentRectForGlyphAtIndex:effectiveRange:

– (NSRect)**lineFragmentRectForGlyphAtIndex:**(unsigned int)*glyphIndex* **effectiveRange:** (NSRange *)*lineFragmentRange*

Returns the line fragment rectangle containing the glyph at *glyphIndex*. The rectangle is defined in the coordinate system of its NSTextContainer. If non-NULL, *lineFragmentRange* is set to contain the range for all glyphs in that line fragment.

Performs glyph generation and layout if needed.

See also: – **lineFragmentUsedRectForGlyphAtIndex:effectiveRange:**, – **setLineFragmentRect: forGlyphRange:usedRect:**

lineFragmentUsedRectForGlyphAtIndex:effectiveRange:

– (NSRect)**lineFragmentUsedRectForGlyphAtIndex:**(unsigned int)*glyphIndex* **effectiveRange:** (NSRange *)*lineFragmentRange*

Returns the portion of the line fragment rectangle containing *glyphAtIndex* that actually contains glyphs (such as for a partial or wrapped line), plus the line fragment padding defined by the NSTextContainer where the glyphs reside. This rectangle is defined in the coordinate system of its NSTextContainer, and is based on line calculation only—that is, it isn't a bounding box for the glyphs in the line fragment.

If non-NULL, *lineFragmentRange* is set to contain the range for all glyphs in the line fragment.

Performs glyph generation and layout if needed.

See also: – **lineFragmentRectForGlyphAtIndex:effectiveRange:**, – **setLineFragmentRect: forGlyphRange:usedRect:**

locationForGlyphAtIndex:

– (NSPoint)**locationForGlyphAtIndex:**(unsigned int)*glyphIndex*

Returns the location, in terms of its line fragment rectangle, for the glyph at *glyphIndex*. The line fragment rectangle in turn is defined in the coordinate system of the text container where it resides.

Performs glyph generation and layout if needed.

See also: – **lineFragmentRectForGlyphAtIndex:effectiveRange:**,
– **lineFragmentUsedRectForGlyphAtIndex:effectiveRange:**

notShownAttributeForGlyphAtIndex:

– (BOOL)**notShownAttributeForGlyphAtIndex:**(unsigned int)*glyphIndex*

Returns YES if the glyph at *glyphIndex* isn't shown (in the sense of the PostScript **show** operator), NO if it is. For example, a tab, newline, or attachment glyph doesn't get shown; it just affects the layout of following glyphs or locates the attachment graphic. Space characters, however, typically are shown as glyphs with a displacement, though they leave no visible marks. Raises an NSRangeException if *glyphIndex* is out of bounds.

Performs glyph generation and layout if needed.

See also: – **setNotShownAttribute:forGlyphAtIndex:**

numberOfGlyphs

– (unsigned int)**numberOfGlyphs**

Returns the number of glyphs in the receiver, performing glyph generation if needed to determine this number.

rangeOfNominallySpacedGlyphsContainingIndex:

– (NSRange)**rangeOfNominallySpacedGlyphsContainingIndex:**(unsigned int)*glyphIndex*

Returns the range for the glyphs around *glyphIndex* that can be displayed with a single PostScript **show** operation; in other words, glyphs with no pairwise kerning or other adjustments to spacing.

Performs glyph generation and layout if needed.

rectArrayForCharacterRange:withinSelectedCharacterRange:inTextContainer:rectCount:

– (NSRect *)**rectArrayForCharacterRange:**(NSRange)*charRange*
withinSelectedCharacterRange:(NSRange)*selCharRange*
inTextContainer:(NSTextContainer *)*aTextContainer*
rectCount:(unsigned int *)*rectCount*

Returns a C array of rectangles for the glyphs in *aTextContainer* that correspond to *charRange*, and by reference in *rectCount* the number of such rectangles. These rectangles can be used to draw the background or highlight for the given range of characters. *selCharRange* indicates selected characters, which can affect the size of the rectangles; it must be equal to or contain *charRange*. To calculate the rectangles for drawing the background, use a selected character range whose location is NSNotFound. To calculate the rectangles for drawing highlighting for *charRange*, use a selected character range that contains *charRange*.

The number of rectangles returned isn't necessarily the number of lines enclosing the specified range. Contiguous lines can share an enclosing rectangle, and lines broken into several fragments have a separate enclosing rectangle for each fragment.

The array of rectangles returned is owned by the receiver, and is overwritten by various NSLayoutManager methods. You should never free it, and should copy it if you need to keep the values or use them after sending other messages to the layout manager.

The purpose of this method is to calculate line rectangles for drawing the text background and highlighting. These rectangles don't necessarily enclose glyphs that draw outside their line fragment rectangles; use **boundingRectForGlyphRange:inTextContainer:** to determine the area that contains all drawing performed for a range of glyphs.

Performs glyph generation and layout if needed.

See also: – **glyphRangeForTextContainer:**, – **characterRangeForGlyphRange:actualGlyphRange:**, – **drawsOutsideLineFragmentForGlyphAtIndex:**

rectArrayForGlyphRange:withinSelectedGlyphRange:inTextContainer:rectCount:

– (NSRect *)**rectArrayForGlyphRange:**(NSRange)*glyphRange*
withinSelectedGlyphRange:(NSRange)*selGlyphRange*
inTextContainer:(NSTextContainer *)*aTextContainer*
rectCount:(unsigned *)*rectCount*

Returns a C array of rectangles for the glyphs in *aTextContainer* in *glyphRange*, and by reference in *rectCount* the number of such rectangles. These rectangles can be used to draw the background or highlight for the given range of glyphs. *selGlyphRange* indicates selected glyphs. To calculate the rectangles for drawing the background, use a selected glyph range whose location is NSNotFound. To calculate the rectangles for highlighting, use a selected glyph range that contains *glyphRange*.

The number of rectangles returned isn't necessarily the number of lines enclosing the specified range. Contiguous lines can share an enclosing rectangle, and lines broken into several fragments have a separate enclosing rectangle for each fragment.

The array of rectangles returned is owned by the receiver, and is overwritten by various NSLayoutManager methods. You should never free it, and should copy it if you need to keep the values or use them after sending other messages to the layout manager.

The purpose of this method is to calculate line rectangles for drawing the text background and highlighting. These rectangles don't necessarily enclose glyphs that draw outside their line fragment rectangles; use **boundingRectForGlyphRange:inTextContainer:** to determine the area that contains all drawing performed for a range of glyphs.

Performs glyph generation and layout if needed.

See also: – **glyphRangeForTextContainer:**, – **drawsOutsideLineFragmentForGlyphAtIndex:**

removeTextContainerAtIndex:

– (void)**removeTextContainerAtIndex:**(unsigned int)*index*

Removes the NSTextContainer at *index* and invalidates the layout as needed.Also invalidates glyph information as needed.

See also: – **addTextContainer:**, – **insertTextContainer:atIndex:**, – **textContainers**,
– **invalidateGlyphsForCharacterRange:changeInLength:actualCharacterRange:**,
– **invalidateLayoutForCharacterRange:isSoft:actualCharacterRange:**

replaceGlyphAtIndex:withGlyph:

– (void)**replaceGlyphAtIndex:**(unsigned int)*glyphIndex* **withGlyph:**(NSGlyph)*newGlyph*

Replaces the glyph at *glyphIndex* with *newGlyph*. Doesn't alter the glyph-to-character mapping or invalidate layout information.

This method is for use by the glyph generation mechanism, and doesn't perform any invalidation or generation of the glyphs or layout. You should never directly invoke this method.

See also: – **setCharacterIndex:forGlyphAtIndex:**, – **invalidateGlyphsForCharacterRange: changeInLength:actualCharacterRange:**, – **invalidateLayoutForCharacterRange:isSoft: actualCharacterRange:**

replaceTextStorage:

– (void)**replaceTextStorage:**(NSTextStorage *)*newTextStorage*

Replaces the NSTextStorage for the group of text-system objects containing the receiver with *newTextStorage*. All NSLayoutManagers sharing the original NSTextStorage then share the new one. This method makes all the adjustments necessary to keep these relationships intact, unlike **setTextStorage:**.

rulerAccessoryViewForTextView:paragraphStyle:ruler:enabled:

– (NSView *)**rulerAccessoryViewForTextView:**(NSTextView *)*aTextView*
paragraphStyle:(NSParagraphStyle *)*paraStyle*
ruler:(NSRulerView *)*aRulerView*
enabled:(BOOL)*flag*

Returns the accessory NSView for *aRulerView*. This accessory contains tab wells, text alignment buttons, and so on. *paraStyle* is used to set the state of the controls in the accessory NSView; it must not be **nil**. If *flag* is YES the accessory view is enabled and accepts mouse and keyboard events; if NO it's disabled.

This method is invoked automatically by the NSTextView object using the layout manager. You should rarely need to invoke it, but you can override it to customize ruler support. If you do this method directly, not that it neither installs the ruler accessory view nor sets the markers for the NSRulerView. You must install the accessory view into the ruler using NSRulerView's **setAccessoryView:** method. To set the markers, use **rulerMarkersForTextView:paragraphSyle:ruler:** to get the markers needed and then send **setMarkers:** to the ruler.

See also: – **horizontalRulerView** (NSScrollView)

rulerMarkersForTextView:paragraphStyle:ruler:

– (NSArray *)**rulerMarkersForTextView:**(NSTextView *)*aTextView*
paragraphStyle:(NSParagraphStyle *)*paraStyle*
ruler:(NSRulerView *)*aRulerView*

Returns the NSRulerMarkers for *aRulerView* in *aTextView*, based on *paraStyle*. These markers represent such things as left and right margins, first-line indent, and tab stops. You can set these markers immediately with NSRulerView's **setMarkers:** method.

This method is invoked automatically by the NSTextView object using the layout manager. You should rarely need to invoke it; but you can override it to add new kinds of markers or otherwise customize ruler support.

See also: – **rulerAccessoryViewForTextView:paragraphStyle:ruler:enabled:**

setBackgroundLayoutEnabled:

– (void)**setBackgroundLayoutEnabled:**(BOOL)*flag*

Sets according to *flag* whether the receiver generates glyphs and lays them out when the application's run loop is idle.

See also: – **backgroundLayoutEnabled**

setCharacterIndex:forGlyphAtIndex:

– (void)**setCharacterIndex:**(unsigned int)*charIndex* **forGlyphAtIndex:**(unsigned int)*glyphIndex*

Maps the character at *charIndex* to the glyph at *glyphIndex*.

This method is for use by the glyph generation mechanism, and doesn't perform any invalidation or generation of the glyphs or layout. You should never directly invoke this method.

See also: – **characterIndexForGlyphAtIndex:**, – **characterRangeForGlyphRange:actualGlyphRange:**, – **glyphRangeForCharacterRange:actualCharacterRange:**

setDelegate:

– (void)**setDelegate:**(id)*anObject*

Sets the receiver's delegate to *anObject*, without retaining it.

See also: – **delegate**

setDrawsOutsideLineFragment:forGlyphAtIndex:

– (void)**setDrawsOutsideLineFragment:**(BOOL)*flag* **forGlyphAtIndex:**(unsigned int)*glyphIndex*

Sets according to *flag* whether the glyph at *glyphIndex* exceeds the bounds of the line fragment where it's laid out. This can happen when text is set at a fixed line height. For example, if the user specifies a fixed line height of 12 points and sets the font size to 24 points, the glyphs will exceed their layout rectangles.

This method is used by the layout mechanism; you should never invoke it directly.

See also: – **drawsOutsideLineFragmentForGlyphAtIndex:**

setExtraLineFragmentRect:usedRect:textContainer:

– (void)**setExtraLineFragmentRect:**(NSRect)*aRect*
usedRect:(NSRect)*usedRect*
textContainer:(NSTextContainer *)*aTextContainer*

Sets a line fragment rectangle for displaying an empty last line in a body of text. *aRect* is the rectangle to set, and *aTextContainer* is the NSTextContainer where the rectangle should be laid out. *usedRect* indicates where the insertion point is drawn.

This method is used by the layout mechanism; you should never invoke it directly.

See also: – **extraLineFragmentRect**, – **extraLineFragmentUsedRect**, – **textContainers**

setHyphenationFactor:

– (void)**setHyphenationFactor:**(float)*factor*

<forthcoming>

See also: – hyphenationFactor

setIntAttribute:value:forGlyphAtIndex:

– (void)**setIntAttribute:**(int)*attributeTag*
value:(int)*anInt*
forGlyphAtIndex:(unsigned int)*glyphIndex*

Sets a custom attribute value for the glyph at *glyphIndex*. *attributeTag* identifies the custom attribute, and *anInt* is its new value.

Subclasses that define their own custom attributes must override this method and provide their own storage for the attribute values. Non-negative tags are reserved by Apple; you can define your own attributes with negative tags and set values using this method.

This method doesn't perform glyph generation or layout. The glyph at *glyphIndex* must already have been generated.

See also: – **intAttribute:forGlyphAtIndex:**

setLineFragmentRect:forGlyphRange:usedRect:

– (void)**setLineFragmentRect:**(NSRect)*fragmentRect*
forGlyphRange:(NSRange)*glyphRange*
usedRect:(NSRect)*usedRect*

Sets to *fragmentRect* the line fragment rectangle where the glyphs in *glyphRange* are laid out. The text container must be specified first with **setTextContainer:forGlyphRange:**, and the exact positions of the glyphs must be set after the line fragment rectangle with **setLocation:forStartOfGlyphRange:**. *usedRect* indicates the portion of *fragmentRect*, in the NSTextContainer's coordinate system, that actually contains glyphs or other marks that are drawn (including the text container's line fragment padding). *usedRect* must be equal to or contained within *fragmentRect*.

This method is used by the layout mechanism; you should never invoke it directly.

See also: – **lineFragmentRectForGlyphAtIndex:effectiveRange:**,
– **lineFragmentUsedRectForGlyphAtIndex:effectiveRange:**

setLocation:forStartOfGlyphRange:

– (void)**setLocation:**(NSPoint)*aPoint* **forStartOfGlyphRange:**(NSRange)*glyphRange*

Sets the location where the glyphs in *glyphRange* are laid out to *aPoint*, which is expressed relative to the origin of the line fragment rectangle for *glyphRange*. *glyphRange* defines a series of glyphs that can be displayed with a single PostScript **show** operation (a nominal range). Setting the location for a series of glyphs implies that the glyphs preceding it can't be included in a single **show** operation.

Before setting the location for a glyph range, you must specify the text container with **setTextContainer:forGlyphRange:** and the line fragment rectangle with **setLineFragmentRect:forGlyphRange:usedRect:**.

This method is used by the layout mechanism; you should never invoke it directly.

See also: – **rangeOfNominallySpacedGlyphsContainingIndex:**

setNotShownAttribute:forGlyphAtIndex:

– (void)**setNotShownAttribute:**(BOOL)*flag* **forGlyphAtIndex:**(unsigned int)*glyphIndex*

Sets according to *flag* whether the glyph at *glyphIndex* is one that isn't shown. For example, a tab or newline character doesn't leave any marks; it just indicates where following glyphs are laid out. Raises an NSRangeException if *glyphIndex* is out of bounds.

This method is used by the layout mechanism; you should never invoke it directly.

See also: – **notShownAttributeForGlyphAtIndex:**

setShowsControlCharacters:

– (void)**setShowsControlCharacters:**(BOOL)*flag*

Controls whether the receiver makes control characters visible in layout where possible. If *flag* is YES, it substitutes visible glyphs for control characters if the font and script support it; if *flag* is NO it doesn't.

See also: – **setShowsInvisibleCharacters:**, – **showsControlCharacters**

setShowsInvisibleCharacters:

– (void)**setShowsInvisibleCharacters:**(BOOL)*flag*

Controls whether the receiver makes whitespace and other typically nonvisible characters visible in layout where possible. If *flag* is YES, it substitutes visible glyphs for invisible characters if the font and script support it; if *flag* is NO it doesn't.

See also: – **setShowsControlCharacters:**, – **showsInvisibleCharacters**

setTextContainer:forGlyphRange:

– (void)**setTextContainer:**(NSTextContainer *)*aTextContainer* **forGlyphRange:** (NSRange)*glyphRange*

Sets to *aTextContainer* the NSTextContainer where the glyphs in *glyphRange* are laid out. You specify the layout within the container with the **setLineFragmentRect:forGlyphRange:usedRect:** and **setLocation: forStartOfGlyphRange:** methods.

This method is used by the layout mechanism; you should never invoke it directly.

See also: – **textContainerForGlyphAtIndex:effectiveRange:**

setTextStorage:

– (void)**setTextStorage:**(NSTextStorage *)*textStorage*

Sets the receiver's NSTextStorage to *textStorage*. This method is invoked automatically when you add an NSLayoutManager to an NSTextStorage object; you should never need to invoke it directly, but might want to override it. If you want to replace the NSTextStorage for an established group of text-system objects containing the receiver, use **replaceTextStorage:**.

See also: – **addLayoutManager:** (NSTextStorage)

setUsesScreenFonts:

– (void)**setUsesScreenFonts:**(BOOL)*flag*

Sets according to *flag* whether the receiver calculates layout and displays text using screen fonts when possible.

See also: – **usesScreenFonts**, – **substituteFontForFont:**

showsControlCharacters

– (BOOL)**showsControlCharacters**

Returns YES if the receiver substitutes visible glyphs for control characters if the font and script support it, NO if it doesn't.

See also: – **showsInvisibleCharacters**, – **setShowsControlCharacters:**

showsInvisibleCharacters

– (BOOL)**showsInvisibleCharacters**

Returns YES if the receiver substitutes visible glyphs for invisible characters if the font and script support it, NO if it doesn't.

See also: – **showsControlCharacters**, – **setShowsInvisibleCharacters:**

substituteFontForFont:

– (NSFont *)**substituteFontForFont:**(NSFont *)*originalFont*

Returns a screen font suitable for use in place of *originalFont*, or simply returns *originalFont* if a screen font can't be used or isn't available. A screen font can be substituted if the receiver is set to use screen fonts and if no NSTextView associated with the receiver are scaled or rotated.

See also: – **usesScreenFonts**

textContainerChangedGeometry:

– (void)**textContainerChangedGeometry:**(NSTextContainer *)*aTextContainer*

Invalidates the layout information, and possibly glyphs, for *aTextContainer* and all subsequent NSTextContainers. This method is invoked automatically by other components of the text system; you should rarely need to invoke it directly. Subclasses of NSTextContainer, however, must invoke this method any time their size of shape changes (a text container that dynamically adjusts its shape to wrap text around placed graphics, for example, must do so when a graphic is added, moved, or removed).

textContainerChangedTextView:

– (void)**textContainerChangedTextView:**(NSTextContainer *)*aTextContainer*

Updates information needed to manage NSTextView objects. This method is invoked automatically by other components of the text system; you should rarely need to invoke it directly.

textContainerForGlyphAtIndex:effectiveRange:

– (NSTextContainer *)**textContainerForGlyphAtIndex:**(unsigned int)*glyphIndex* **effectiveRange:** (NSRange *)*effectiveGlyphRange*

Returns the NSTextContainer where the glyph at *glyphIndex* is laid out. If non-NULL, *effectiveGlyphRange* is set to the range for all glyphs laid out in that text container.

Performs glyph generation and layout if needed.

See also: – **setTextContainer:forGlyphRange:**

textContainers

– (NSArray *)**textContainers**

Returns the receiver's NSTextContainers.

See also: – **addTextContainer:**, – **insertTextContainer:atIndex:**, – **removeTextContainerAtIndex:**

textStorage

– (NSTextStorage *)**textStorage**

Returns the receiver's NSTextStorage.

See also: – **setTextStorage:**, – **replaceTextStorage:**

textStorage:edited:range:changeInLength:invalidatedRange:

– (void)**textStorage:**(NSTextStorage *)*aTextStorage*
edited:(unsigned int)*mask*
range:(NSRange)*range*
changeInLength:(int)*lengthChange*
invalidatedRange:(NSRange)*invalidatedCharRange*

Invalidates glyph and layout information for a portion of text in *aTextStorage*. This message is sent from NSTextStorage's **processEditing** method to indicate that its characters or attributes have been changed.

This method invalidates glyphs and layout for the affected characters, and performs a soft invalidation of the layout information for all subsequent characters. *mask* specifies the nature of the changes. Its value is made by combining these options with the C bitwise OR operator:

Option	Meaning
NSTextStorageEditedAttributes	Attributes were added, removed, or changed.
NSTextStorageEditedCharacters	Characters were added, removed, or replaced.

range indicates the extent of characters resulting from the edits. If the NSTextStorageEditedCharacters bit of *mask* is set, *lengthChange* gives the number of characters added to or removed from the original range (otherwise its value is irrelevant). For example, after replacing “The” with “Several” to produce the string “Several files couldn’t be saved”, *range* is {0, 7} and *lengthChange* is 4. The receiver uses this information to update its character-to-glyph mapping and to update the selection range based on the change.

invalidatedRange represents the range of characters affected after attributes have been fixed. For example, deleting a paragraph separator character invalidates the layout information for all characters in the paragraphs that precede and follow the separator.

textStorage:edited:range:changeInLength:invalidatedRange: messages are sent in a series to each NSLayoutManager associated with the text storage object, so the NSLayoutManagers receiving them shouldn’t edit *aTextStorage*. If one of them does, the *range*, *lengthChange*, and *invalidatedRange* arguments will be incorrect for all following NSLayoutManagers that receive the message.

See also: – **invalidateLayoutForCharacterRange:isSoft:actualCharacterRange:**

textViewForBeginningOfSelection

– (NSTextView *)**textViewForBeginningOfSelection**

Returns the NSTextView containing the first glyph in the selection, or **nil** if there’s no selection or if there isn’t enough layout information to determine the text view.

underlineGlyphRange:underlineType:lineFragmentRect:lineFragmentGlyphRange:containerOrigin:

– (void)**underlineGlyphRange:**(NSRange)*glyphRange*
underlineType:(int)*underlineType*
lineFragmentRect:(NSRect)*lineRect*
lineFragmentGlyphRange:(NSRange)*lineGlyphRange*
containerOrigin:(NSPoint)*containerOrigin*

Calculates and draws underlining for the glyphs in *glyphRange*, which must belong to a single line fragment rectangle (as returned by **lineFragmentRectForGlyphAtIndex:effectiveRange:**). *underlineType* indicates the style of underlining to draw; NSLayoutManager accepts only NSSingleUnderlineStyle, but subclasses can define their own underline styles. *lineRect* is the line fragment rectangle containing the glyphs to draw underlining for, and *lineGlyphRange* is the range of all glyphs within that line fragment rectangle. *containerOrigin* is the origin of the line fragment rectangle's NSTextContainer in its NSTextView.

This method determines which glyphs actually need to be underlined based on *underlineType*. With NSSingleUnderlineStyle, for example, leading and trailing whitespace isn't underlined, but whitespace between visible glyphs is. A potential word-underline style would omit underlining on any whitespace. After determining which glyphs to draw underlining on, this method invokes **drawUnderlineForGlyphRange:...** for each contiguous range of glyphs that requires it.

See also: – **textContainerForGlyphAtIndex:effectiveRange:**, – **textContainerOrigin** (NSTextView)

usedRectForTextContainer:

– (NSRect)**usedRectForTextContainer:**(NSTextContainer *)*aTextContainer*

Returns the bounding rectangle for the glyphs laid out in *aTextContainer*, which tells "how full" it is. This rectangle is given in the *aTextContainer*'s coordinate system.

See also: – **containerSize** (NSTextContainer)

usesScreenFonts

– (BOOL)**usesScreenFonts**

Returns YES if the receiver calculates layout and displays text using screen fonts when possible, NO otherwise.

See also: – **setUsesScreenFonts:**, – **substituteFontForFont:**

Methods Implemented By the Delegate

layoutManager:didCompleteLayoutForTextContainer:atEnd:

– (void)**layoutManager:**(NSLayoutManager *)*aLayoutManager*
didCompleteLayoutForTextContainer:(NSTextContainer *)*aTextContainer*
atEnd:(BOOL)*flag*

Informs the delegate that *aLayoutManager* has finished laying out text in *aTextContainer*. *aTextContainer* is **nil** if there aren't enough containers to hold all the text; the delegate can use this information as a cue to add another container. If *flag* is YES, *aLayoutManager* is finished laying out its text—this also means that *aTextContainer* is the final text container used by the layout manager. Delegates can use this information to show an indicator or background or to enable or disable a button that forces immediate layout of text.

layoutManagerDidInvalidateLayout:

– (void)**layoutManagerDidInvalidateLayout:**(NSLayoutManager *)*aLayoutManager*

Informs the delegate that *aLayoutManager* has invalidated layout information (not glyph information). This method is invoked only when layout was complete and then became invalidated for some reason. Delegates can use this information to show an indicator or background layout or to enable a button that forces immediate layout of text.

NSMatrix

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

Class Description

NSMatrix is a class used for creating groups of NSCells that work together in various ways. It includes methods for arranging NSCells in rows and columns, either with or without space between them. NSCells in an NSMatrix are numbered by row and column, each starting with 0; for example, the top left NSCell would be at (0, 0), and the NSCell that's second down and third across would be at (1, 2).

The cell objects that an NSMatrix contains are usually of a single subclass of NSCell, but they can be of multiple subclasses of NSCell. The only restriction is that all cell objects must be the same size. An NSMatrix can be set up to create new NSCells by copying a prototype object, or by allocating and initializing instances of a specific NSCell class. Cells created by or added to an NSMatrix are retained by the matrix.

An NSMatrix adds to NSControl's target/action paradigm by allowing a separate target and action for each of its NSCells in addition to its own target and action. It also allows for an action message that's sent when the user double-clicks an NSCell, which is sent in addition to the single-click action message. If an NSCell doesn't have an action, the NSMatrix sends its own action to its own target. If an NSCell doesn't have a target, the NSMatrix sends the NSCell's action to its own target. The double-click action of an NSMatrix is always sent to the target of the NSMatrix.

Since the user might press the mouse button while the cursor is within the NSMatrix and then drag the mouse around, NSMatrix offers four "selection modes" that determine how NSCells behave when the NSMatrix is tracking the mouse:

- NSTrackModeMatrix is the most basic mode of operation. In this mode the NSCells are asked to track the mouse with **trackMouse:inRect:ofView:untilMouseUp:** whenever the mouse is inside their bounds. No highlighting is performed. An example of this mode might be a "graphic equalizer" NSMatrix of sliders, where moving the mouse around causes the sliders to move under the mouse.
- NSHighlightModeMatrix is a modification of NSTrackModeMatrix. In this mode, an NSCell is highlighted before it's asked to track the mouse, then unhighlighted when it's done tracking. This is useful for multiple unconnected NSCells that use highlighting to inform the user that they are being tracked (like push-buttons and switches).

- NSRadioModeMatrix is used when you want no more than one NSCell to be selected at a time. It can be used to create a set of buttons of which one and only one is selected (there's the option of allowing no button to be selected). Any time an NSCell is selected, the previously selected NSCell is unselected. The canonical example of this mode is a set of radio buttons.
- NSListModeMatrix is the opposite of NSTrackModeMatrix. NSCells are highlighted, but don't track the mouse. This mode can be used to select a range of text values, for example. NSMatrix supports the standard multiple-selection paradigms of dragging to select, using the shift key to make discontinuous selections, and using the alternate key to extend selections. Browsers (as used, for instance, in NeXT's File Viewer) use this mode.

Method Types

Initializing an NSMatrix object

– initWithFrame:
– initWithFrame:mode:cellClass:numberOfRows: numberOfColumns:
– initWithFrame:mode:prototype:numberOfRows: numberOfColumns:

Setting the selection mode

– mode
– setMode:

Configuring the NSMatrix

– allowsEmptySelection
– isSelectionByRect
– setAllowsEmptySelection:
– setSelectionByRect:

Setting the cell class

– cellClass
– prototype
– setCellClass:
– setPrototype:

Laying out the NSMatrix

– addColumn
– addColumnWithCells:
– addRow
– addRowWithCells:
– cellFrameAtRow:column:
– cellSize
– getNumberOfRows:columns:
– insertColumn:
– insertColumn:withCells:
– insertRow:
– insertRow:withCells:
– intercellSpacing
– makeCellAtRow:column:
– numberOfColumns
– numberOfRows
– putCell:atRow:column:
– removeColumn:
– removeRow:
– renewRows:columns:
– setCellSize:
– setIntercellSpacing:
– sortUsingFunction:context:
– sortUsingSelector:

Finding matrix coordinates

– getRow:column:forPoint:
– getRow:column:ofCell:

Modifying individual cells

– setState:atRow:column:

Selecting cells

– deselectAllCells
– deselectSelectedCell
– keyCell
– selectAll:
– selectCellAtRow:column:
– selectCellWithTag:
– selectedCell
– selectedCells
– selectedColumn
– selectedRow
– setKeyCell:
– setSelectionFrom:to:anchor:highlight:

Finding cells

– cellAtRow:column:
– cellWithTag:
– cells

Modifying graphic attributes

– backgroundColor
– cellBackgroundColor
– drawsBackground
– drawsCellBackground
– setBackgroundColor:
– setCellBackgroundColor:
– setDrawsBackground:
– setDrawsCellBackground:

Editing text in cells

– selectText:
– selectTextAtRow:column:
– textDidBeginEditing:
– textDidChange:
– textDidEndEditing:
– textShouldBeginEditing:
– textShouldEndEditing:

Setting tab key behavior

– nextText
– previousText
– setNextText:
– setPreviousText:
– setTabKeyTraversesCells:
– tabKeyTraversesCells

Assigning a delegate

– delegate
– setDelegate:

Resizing the matrix and its cells

– autosizesCells
– setAutosizesCells:
– setValidateSize:
– sizeToCells

Scrolling

– isAutoscroll
– scrollCellToVisibleAtRow:column:
– setAutoscroll:
– setScrollable:

Displaying

– drawCellAtRow:column:
– highlightCell:atRow:column:

Target and action

– doubleAction
– errorAction
– sendAction
– sendAction:to:forAllCells:
– sendDoubleAction
– setDoubleAction:
– setErrorAction:

Handling event and action messages

– acceptsFirstMouse:
– mouseDown:
– mouseDownFlags
– performKeyEquivalent:

Managing the cursor

– resetCursorRects

Instance Methods

acceptsFirstMouse:

– (BOOL)**acceptsFirstMouse:**(NSEvent *)*theEvent*

Returns NO if the selection mode of the NSMatrix is NSListModeMatrix, YES if the NSMatrix is in any other selection mode. The NSMatrix does not accept first mouse in NSListModeMatrix to prevent the loss of multiple selections. The NSEvent parameter, *theEvent*, is ignored.

See also: – **mode**

addColumn

– (void)**addColumn**

Adds a new column of cells to the right of the last column, creating new cells as needed with **makeCellAtRow:column:**.

If the number of rows or columns in the NSMatrix has been changed with **renewRows:columns:**, new cells are created only if they are needed. This allows you to grow and shrink an NSMatrix without repeatedly creating and freeing the cells.

This method doesn't redraw the NSMatrix. Accordingly, after calling this method you should send **setNeedsDisplay:YES** to the NSMatrix. Your code may also need to use **sizeToCells** after sending this method to resize the NSMatrix to fit the newly added cells.

See also: – **cellClass**, – **insertColumn:**, – **prototype**

addColumnWithCells:

– (void)**addColumnWithCells:**(NSArray *)*newCells*

Adds a new column of cells to the right of the last column. The new column is filled with objects from *newCells*, starting with the object at index 0. Each object in *newCells* should be a an NSCell or one of its subclasses (usually NSActionCell). *newCells* should have a sufficient number of cells to fill the entire column. Extra cells are ignored, unless the matrix is empty. In that case, a matrix is created with one column and enough rows for all the elements of *newCells*.

This method doesn't redraw the NSMatrix. Accordingly, after calling this method you should send **setNeedsDisplay:YES** to the NSMatrix. Your code may also need to use **sizeToCells** after sending this method to resize the NSMatrix to fit the newly added cells.

See also: – **insertColumn:withCells:**

addRow

– (void)**addRow**

Adds a new row of cells below the last row, creating new cells as needed with **makeCellAtRow:column:**.

If the number of rows or columns in the NSMatrix has been changed with **renewRows:columns:**, then new cells are created only if they are needed. This allows you to grow and shrink an NSMatrix without repeatedly creating and freeing the cells.

This method doesn't redraw the NSMatrix. Accordingly, after calling this method you should send **setNeedsDisplay:YES** to the NSMatrix. Your code may also need to use **sizeToCells** after sending this method to resize the NSMatrix to fit the newly added cells.

See also: – **cellClass**, – **insertRow:**, – **prototype**

addRowWithCells:

– (void)**addRowWithCells:**(NSArray *)*newCells*

Adds a new row of cells below the last row. The new row is filled with objects from *newCells*, starting with the object at index 0. Each object in *newCells* should be a an NSCell or one of its subclasses (usually NSActionCell). *newCells* should have a sufficient number of cells to fill the entire row. Extra cells are

ignored, unless the matrix is empty. In that case, a matrix is created with one row and enough columns for all the elements of *newCells*.

This method doesn't redraw the NSMatrix. Accordingly, after calling this method you should send **setNeedsDisplay:YES** to the NSMatrix. Your code may also need to use **sizeToCells** after sending this method to resize the NSMatrix to fit the newly added cells.

See also: – **insertRow:withCells:**

allowsEmptySelection

– (BOOL)**allowsEmptySelection**

Returns whether its possible to have no cells selected in a radio-mode matrix.

See also: – **mode**

autosizesCells

– (BOOL)**autosizesCells**

Returns YES if cells are resized proportionally to the NSMatrix when its size changes (and inter-cell spacing is kept constant). Returns NO if the cell size remains constant (and inter-cell spacing changes).

backgroundColor

– (NSColor *)**backgroundColor**

Returns the color used to draw the background of the NSMatrix (the space between the cells).

See also: – **cellBackgroundColor**, – **drawsBackground**

cellAtRow:column:

– (id)**cellAtRow:**(int)*row* **column:**(int)*column*

Returns the NSCell object at *row* and *column*, or **nil** if either *row* or *column* are outside the bounds of the NSMatrix.

See also: – **getRow:column:ofCell:**

cellBackgroundColor

– (NSColor *)**cellBackgroundColor**

Returns the color used to fill the background of the NSMatrix's cells.

See also: – **backgroundColor**, – **drawsCellBackground**

cellClass

– (Class)**cellClass**

Returns the subclass of NSCell that the NSMatrix uses when creating new (empty) cells.

See also: – **prototype**, – **makeCellAtRow:column:**

cellFrameAtRow:column:

– (NSRect)**cellFrameAtRow:**(int)*row* **column:**(int)*column*

Returns the frame rectangle of the cell that would be drawn at the specified *row* and *column* (whether or not the specified cell actually exists).

See also: – **cellSize**

cellSize

– (NSSize)**cellSize**

Returns the width and the height of each cell in the NSMatrix (all cells in an NSMatrix are the same size).

See also: – **cellFrameAtRow:column**:, – **intercellSpacing**

cellWithTag:

– (id)**cellWithTag:**(int)*anInt*

Searches the NSMatrix and returns the last (when viewing the matrix as a row-ordered array) NSCell object which has a tag matching *anInt*, or **nil** if no such cell exists.

See also: – **selectCellWithTag:**, – **setTag:** (NSActionCell)

cells

– (NSArray *)**cells**

Returns an NSArray that contains the NSMatrix's cells. The cells in the array are row-ordered; that is, the first row of cells appear first in the array, followed by the second row, and so forth.

See also: – **cellAtRow:column:**

delegate

– (id)**delegate**

Returns the delegate for messages from the field editor.

See also: – **textShouldBeginEditing:**, – **textShouldEndEditing:**

deselectAllCells

– (void)**deselectAllCells**

Deselects all cells in the NSMatrix and, if necessary, redisplays the NSMatrix. If the selection mode is NSRadioModeMatrix and empty selection is not allowed, this method does nothing.

See also: – **allowsEmptySelection**, – **mode**, – **selectAll:**

deselectSelectedCell

– (void)**deselectSelectedCell**

Deselects the selected cell or cells. If the selection mode is NSRadioModeMatrix and empty selection is not allowed, or if nothing is currently selected, this method does nothing. This method doesn't redisplay the NSMatrix.

See also: – **allowsEmptySelection**, – **mode**, – **selectCellAtRow:column:**

doubleAction

– (SEL)**doubleAction**

Returns the action method sent by the NSMatrix to its target when the user double-clicks an entry, or NULL if there's no double-click action. The double-click action of an NSMatrix is sent after the appropriate single-click action (for the NSCell clicked or for the NSMatrix if the NSCell doesn't have its own action).

If there is no double-click action and the NSMatrix doesn't ignore multiple clicks, the single-click action is sent twice.

See also: – **action** (NSControl), – **target** (NSControl), – **sendDoubleAction**, – **ignoresMultiClick** (NSControl)

drawCellAtRow:column:

– (void)**drawCellAtRow:**(int)*row* **column:**(int)*column*

Displays the cell at the specified row and column, providing that *row* and *column* reference a cell that's within the NSMatrix.

See also: – **drawCell:** (NSControl), – **drawCellInside:** (NSControl)

drawsBackground

– (BOOL)**drawsBackground**

Returns whether the receiver draws its background (the space between the cells).

See also: – **backgroundColor**, – **drawsCellBackground**

drawsCellBackground

– (BOOL)**drawsCellBackground**

Returns whether the receiver draws the background within each of its cells.

See also: – **cellBackgroundColor**, – **drawsBackground**

errorAction

– (SEL)**errorAction**

Returns the action that's sent to the target of the NSMatrix when the user enters an illegal value for the selected cell.

See also: – **action** (NSControl), – **target** (NSControl), – **textShouldEndEditing:**

getNumberOfRows:columns:

– (void)**getNumberOfRows:**(int *)*rowCount* **columns:**(int *)*columnCount*

Returns by reference the number of rows and columns in the NSMatrix.

See also: – **numberOfColumns**, – **numberOfRows**

getRow:column:forPoint:

– (BOOL)**getRow:**(int *)*row* **column:**(int *)*column* **forPoint:**(NSPoint)*aPoint*

Returns YES if *aPoint* lies within one of the cells in the NSMatrix, and returns by reference the row and column for the cell within which the specified point lies. If *aPoint* falls outside the bounds of the Matrix or lies within an intercell spacing, **getRow:column:forPoint:** returns NO.

Make sure that *aPoint* is in the coordinate system of the NSMatrix.

See also: – **getRow:column:ofCell:**

getRow:column:ofCell:

– (BOOL)**getRow:**(int *)*row* **column:**(int *)*column* **ofCell:**(NSCell *)*aCell*

Searches the NSMatrix and returns YES if *aCell* is one of the cells in the NSMatrix, and returns by reference the row and column of the cell. If *aCell* is not found within the Matrix, **getRow:column:ofCell:** returns NO.

See also: – **getRow:column:forPoint:**

highlightCell:atRow:column:

– (void)**highlightCell:**(BOOL)*flag* **atRow:**(int)*row* **column:**(int)*column*

Assuming that *row* and *column* indicate a valid cell within the NSMatrix, **highlightCell:atRow:column:** highlights (if *flag* is YES) or unhighlights (if *flag* is NO) the specified cell.

initWithFrame:

– (id)**initWithFrame:**(NSRect)*frameRect*

Initializes and returns the receiver, a newly-allocated instance of NSMatrix, with default parameters in the frame specified by *frameRect*. The new NSMatrix contains no rows or columns. The default mode is NSRadioModeMatrix. The default cell class is NSActionCell.

initWithFrame:mode:cellClass:numberOfRows:numberOfColumns:

– (id)**initWithFrame:**(NSRect)*frameRect*
mode:(int)*aMode*
cellClass:(Class)*factoryId*
numberOfRows:(int)*numRows*
numberOfColumns:(int)*numColumns*

Initializes and returns the receiver, a newly-allocated instance of NSMatrix, in the frame specified by *frameRect*. The new NSMatrix contains *numRows* rows and *numColumns* columns. *aMode* is set as the tracking mode for the NSMatrix, and can be one of the following four constants, all of which are described in the class description:

Tracking mode	Description
NSTrackModeMatrix	Cells track the mouse, but do not highlight
NSHighlightModeMatrix	Cells highlight as they track the mouse
NSRadioModeMatrix	Allows no more than one selected cell
NSListModeMatrix	Cells are highlighted, but don't track the mouse

The new NSMatrix creates and uses cells of class *classId*, which can be obtained by sending a **class** message to the desired subclass of NSCell.

This method is the designated initializer for matrices that add cells by creating instances of an NSCell subclass.

initWithFrame:mode:prototype:numberOfRows:numberOfColumns:

– (id)**initWithFrame:**(NSRect)*frameRect* **mode:**(int)*aMode* **prototype:**(NSCell *)*aCell*
numberOfRows:(int)*numRows* **numberOfColumns:**(int)*numColumns*

Initializes and returns the receiver, a newly-allocated instance of NSMatrix, in the frame specified by *frameRect*. The new NSMatrix contains *numRows* rows and *numColumns* columns. *aMode* is set as the tracking mode for the NSMatrix, and can be one of the following four constants, all of which are described in the class description:

Tracking mode	Description
NSTrackModeMatrix	Cells track the mouse, but do not highlight
NSHighlightModeMatrix	Cells highlight as they track the mouse

Tracking mode	Description
NSRadioModeMatrix	Allows no more than one selected cell
NSListModeMatrix	Cells are highlighted, but don't track the mouse

The new Matrix creates cells by copying *aCell*, which should be an instance of a subclass of NSCell.

This method is the designated initializer for matrices that add cells by copying an instance of an NSCell subclass.

insertColumn:

– (void)**insertColumn:**(int)*column*

Inserts a new column of cells before *column*, creating new cells if needed with **makeCellAtRow:column:** . If *column* is greater than the number of columns in the NSMatrix, enough columns are created to expand the NSMatrix to be *column* columns wide. This method doesn't redraw the NSMatrix. Accordingly, after calling this method you should send **setNeedsDisplay:YES** to the NSMatrix. Your code may also need to use **sizeToCells** after sending this method to resize the NSMatrix to fit the newly added cells.

If the number of rows or columns in the NSMatrix has been changed with **renewRows:columns:**, then new cells are created only if they're needed. This allows you to grow and shrink an NSMatrix without repeatedly creating and freeing the cells.

See also: – **addColumn**, – **insertRow:**

insertColumn:withCells:

– (void)**insertColumn:**(int)*column* **withCells:**(NSArray *)*newCells*

Inserts a new column of cells before *column*. The new column is filled with objects from *newCells*, starting with the object at index 0. Each object in *newCells* should be a an NSCell or one of its subclasses (usually NSActionCell). If *column* is greater than the number of columns in the NSMatrix, enough columns are created to expand the NSMatrix to be *column* columns wide. *newCells* should either be empty or contain a sufficient number of cells to fill each new column. If *newCells* is **nil** or an array with no elements, the call is equivalent to calling **insertColumn:**. Extra cells are ignored, unless the matrix is empty. In that case, a matrix is created with one column and enough rows for all the elements of *newCells*.

This method doesn't redraw the NSMatrix. Accordingly, after calling this method you should send **setNeedsDisplay:YES** to the NSMatrix. Your code may also need to use **sizeToCells** after sending this method to resize the NSMatrix to fit the newly added cells.

See also: – **addColumnWithCells:**, – **insertRow:withCells:**

insertRow:

– (void)**insertRow:**(int)*row*

Inserts a new row of cells before *row*, creating new cells if needed with **makeCellAtRow:column:**. If *row* is greater than the number of rows in the NSMatrix, enough rows are created to expand the NSMatrix to be *row* rows high. This method doesn't redraw the NSMatrix. Accordingly, after calling this method you should send **setNeedsDisplay:YES** to the NSMatrix. Your code may also need to use **sizeToCells** after sending this method to resize the NSMatrix to fit the newly added cells.

If the number of rows or columns in the NSMatrix has been changed with **renewRows:columns:**, then new cells are created only if they're needed. This allows you to grow and shrink an NSMatrix without repeatedly creating and freeing the cells.

See also: – **addRow**, – **insertColumn:**

insertRow:withCells:

– (void)**insertRow:**(int)*row* **withCells:**(NSArray *)*newCells*

Inserts a new row of cells before *row*. The new row is filled with objects from *newCells*, starting with the object at index 0. Each object in *newCells* should be an NSCell or one of its subclasses (usually NSActionCell). If *row* is greater than the number of rows in the NSMatrix, enough rows are created to expand the NSMatrix to be *row* rows high. *newCells* should either be empty or contain a sufficient number of cells to fill each new row (**[***newCells* **count]** must be zero or at least **[self numberOfColumns]**). If *newCells* is **nil** or an array with no elements, the call is equivalent to calling **insertRow:**. Extra cells are ignored, unless the matrix is empty. In that case, a matrix is created with one row and enough columns for all the elements of *newCells*.

This method doesn't redraw the NSMatrix. Accordingly, after calling this method you should send **setNeedsDisplay:YES** to the NSMatrix. Your code may also need to use **sizeToCells** after sending this method to resize the NSMatrix to fit the newly added cells.

See also: – **addRowWithCells:**, – **insertColumn:withCells:**

intercellSpacing

– (NSSize)**intercellSpacing**

Returns the vertical and horizontal spacing between cells in the NSMatrix.

See also: – **cellSize**

isAutoscroll

– (BOOL)**isAutoscroll**

Returns whether the NSMatrix will be automatically scrolled whenever the mouse is dragged outside the NSMatrix after a mouse-down event within its bounds.

See also: – **scrollCellToVisibleAtRow:column:**, – **setScrollable:**

isSelectionByRect

– (BOOL)**isSelectionByRect**

Returns YES if a the user can select a rectangle of cells in the NSMatrix by dragging the cursor, NO otherwise.

See also: – **setSelectionFrom:to:anchor:highlight:**

keyCell

– (id)**keyCell**

Returns the cell that will be clicked when the user presses the Return key.

See also: – **nextText**, – **tabKeyTraversesCells**

makeCellAtRow:column:

– (NSCell *)**makeCellAtRow:**(int)*row* **column:**(int)*column*

Creates a new cell at the specified location in the NSMatrix. If the NSMatrix has a prototype cell, it's copied to create the new cell. If not, and if the NSMatrix has a cell class set, it allocates and initializes (with **init**) an instance of that class. If the NSMatrix hasn't had either a prototype cell or a cell class set, **makeCellAtRow:column:** creates an NSActionCell. Returns the newly created cell.

Your code should never invoke this method directly; it's used by **addRow** and other methods when a cell must be created. It may be overridden to provide more specific initialization of cells.

See also: – **addColumn**, – **addRow**, – **insertColumn:**, – **insertRow:**, – **setCellClass:**, – **setPrototype:**

mode

– (NSMatrixMode)**mode**

Returns the selection mode of the NSMatrix. Possible return values are defined in NSMatrix.h, and are also listed here:

Tracking mode	Description
NSTrackModeMatrix	Cells track the mouse, but do not highlight
NSHighlightModeMatrix	Cells highlight as they track the mouse
NSRadioModeMatrix	Allows no more than one selected cell
NSListModeMatrix	Cells are highlighted, but don't track the mouse

These modes are explained in detail in the class description.

See also: – **initWithFrame:mode:cellClass:numberOfRows:numberOfColumns:**, – **initWithFrame:mode:prototype:numberOfRows:numberOfColumns:**

mouseDown:

– (void)**mouseDown:**(NSEvent *)*theEvent*

Responds to a mouse-down event. A mouse-down event in a text cell initiates editing mode. A double-click in any cell type except a text cell sends the double-click action of the NSMatrix (if there is one) in addition to the single-click action.

Your code should never invoke this method, but you may override it to implement different mouse tracking than NSMatrix does. The response of the NSMatrix depends on its selection mode, as explained in the class description.

See also: – **sendAction**, – **sendDoubleAction**

mouseDownFlags

– (int)**mouseDownFlags**

Returns the flags that were in effect at the mouse-down event that started the current tracking session (NSMatrix's **mouseDown:** method obtains these flags by sending a **modifierFlags** message to the event passed into **mouseDown:**). Use this method if you want to access these flags. This method is valid only

during tracking; it isn't useful if the target of the NSMatrix initiates another tracking loop as part of its action method (as a cell that pops up a PopUpList does, for example).

See also: – **sendActionOn:** (NSCell)

nextText

– (id)**nextText**

Returns the object that would be selected if the user presses Tab while editing the last text cell in the NSMatrix.

See also: – **nextKeyView** (NSView), – **previousText**, – **setNextText:**

numberOfColumns

– (int)**numberOfColumns**

Returns the number of columns in the NSMatrix.

See also: – **getNumberOfRows:columns:**

numberOfRows

– (int)**numberOfRows**

Returns the number of rows in the NSMatrix.

See also: – **getNumberOfRows:columns:**

performKeyEquivalent:

– (BOOL)**performKeyEquivalent:**(NSEvent *)*theEvent*

If there's a cell in the NSMatrix that has a key equivalent equal to the character in **[***theEvent* **charactersIgnoringModifiers]** (taking into account any key modifier flags) and that cell is enabled, that cell is made to react as if the user had clicked it: by highlighting, changing its state as appropriate, sending its action if it has one, and then unhighlighting. Returns YES if a cell in the NSMatrix responds to the key equivalent in *theEvent*, NO if no cell responds.

Your code should never send this message; it is sent when the NSMatrix or one of its superviews is the first responder and the user presses a key. You may want to override this method to change the way key equivalents are performed or displayed, or to disable them in your subclass.

previousText

– (id)**previousText**

Returns the object that would be selected if the user presses Shift-Tab while editing the first text cell in the NSMatrix.

See also: – **nextKeyView** (NSView), – **nextText**, – **setPreviousText:**

prototype

– (id)**prototype**

Returns the prototype cell that's copied whenever a new cell needs to be created, or **nil** if there is none.

See also: – **initWithFrame:mode:prototype:numberOfRows:numberOfColumns:**,
– **makeCellAtRow:column:**

putCell:atRow:column:

– (void)**putCell:**(NSCell *)*newCell* **atRow:**(int)*row* **column:**(int)*column*

Replaces the cell at the specified *row* and *column* with *newCell*, and redraws the cell.

removeColumn:

– (void)**removeColumn:**(int)*column*

Removes the column at position *column* from the NSMatrix and autoreleases the column's cells. Doesn't redraw the NSMatrix. Your code should normally send **sizeToCells** after invoking this method to resize the NSMatrix so that it fits the reduced cell count.

See also: – **removeRow:**, – **addColumn**, – **insertColumn:**

removeRow:

– (void)**removeRow:**(int)*row*

Removes the row at position *row* from the NSMatrix and autoreleases the row's cells. Doesn't redraw the NSMatrix. Your code should normally send **sizeToCells** after invoking this method to resize the NSMatrix so that it fits the reduced cell count.

See also: – **removeColumn:**, – **addRow**, – **insertRow:**

renewRows:columns:

– (void)**renewRows:**(int)*newRows* **columns:**(int)*newCols*

Changes the number of rows and columns in the NSMatrix. This method uses the same cells as before, creating new cells only if the new size is larger; it never frees cells. Doesn't redisplay the NSMatrix. Your code should normally send **sizeToCells** after invoking this method to resize the NSMatrix so that it fits the changed cell arrangement. This method deselects all cells in the NSMatrix.

See also: – **addColumn**, – **addRow**, – **removeColumn:**, – **removeRow:**

resetCursorRects

– (void)**resetCursorRects**

Resets cursor rectangles so that the cursor becomes an I-beam over text cells. It does this by sending **resetCursorRect:inView:** to each cell in the NSMatrix. Any cell that has a cursor rectangle to set up should then send **addCursorRect:cursor:** back to the NSMatrix.

See also: – **resetCursorRect:inView:** (NSCell), – **addCursorRect:cursor:** (NSView)

scrollCellToVisibleAtRow:column:

– (void)**scrollCellToVisibleAtRow:**(int)*row* **column:**(int)*column*

If the NSMatrix is in a scrolling view, and *row* and *column* represent a valid cell within the NSMatrix, this method scrolls the NSMatrix so that the specified cell is visible.

See also: – **scrollRectToVisible:** (NSView)

selectAll:

– (void)**selectAll:**(id)*sender*

Selects and highlights all of the cells in the NSMatrix, except for editable text cells and disabled cells. Redisplays the NSMatrix. *sender* is ignored.

See also: – **selectCell:** (NSControl)

selectCellAtRow:column:

– (void)**selectCellAtRow:**(int)*row* **column:**(int)*column*

Selects the cell at the specified *row* and *column* within the NSMatrix. If the specified cell is an editable text cell, its text is selected. If either *row* or *column* is –1, then the current selection is cleared (unless the NSMatrix is in NSRadioModeMatrix and doesn't allow empty selection). Redraws the affected cells.

See also: – **allowsEmptySelection**, – **mode**, – **selectCell:** (NSControl)

selectCellWithTag:

– (BOOL)**selectCellWithTag:**(int)*anInt*

If the NSMatrix has at least one cell whose tag is equal to *anInt*, the last cell (when viewing the matrix as a row-ordered array) is selected. If the specified cell is an editable text Cell, its text is selected. Returns YES if the NSMatrix contains a cell whose tag matches *anInt*, or NO if no such cell exists.

See also: – **cellWithTag:**, – **selectCell:** (NSControl)

selectText:

– (void)**selectText:**(id)*sender*

If the currently selected cell is editable and enabled, its text is selected. Otherwise, the key cell is selected.

See also: – **keyCell**, – **selectText:** (NSTextField)

selectTextAtRow:column:

– (id)**selectTextAtRow:**(int)*row* **column:**(int)*column*

If *row* and *column* indicate a valid cell within the NSMatrix, and that cell is both editable and selectable, **selectTextAtRow:column:** selects and then returns the specified cell. If the cell specified by *row* and *column* is either not editable or not selectable, **selectTextAtRow:column:** does nothing, and returns **nil**. Finally, if *row* and *column* indicate a cell that is outside the NSMatrix, **selectTextAtRow:column:** does nothing and returns the receiver.

See also: – **selectText:**

selectedCell

– (id)**selectedCell**

Returns the most recently selected cell, or **nil** if no cell is selected. If more than one cell is selected, **selectedCell** returns the cell that is lowest and furthest to the right in the NSMatrix.

selectedCells

– (NSArray *)**selectedCells**

Returns an array containing each of the cells in the receiver that is currently highlighted.

See also: – **selectedCell**

selectedColumn

– (int)**selectedColumn**

Returns the column number of the selected cell, or –1 if no cells are selected. If cells in multiple columns are selected, this method returns the number of the last (right-most) column containing a selected cell.

selectedRow

– (int)**selectedRow**

Returns the row number of the selected cell, or –1 if no cells are selected. If cells in multiple rows are selected, this method returns the number of the last row containing a selected cell.

sendAction

– (BOOL)**sendAction**

If the selected cell has both an action and a target, its action is sent to its target. If the cell has an action but no target, its action is sent to the target of the NSMatrix. If the cell doesn't have an action, or if there is no selected cell, the NSMatrix sends its own action to its target. Returns YES if an action was successfully sent to a target.

If the selected cell is disabled, this method does nothing and returns NO.

See also: – **sendDoubleAction**, – **action** (NSCell), – **target** (NSCell)

sendAction:to:forAllCells:

– (void)**sendAction:**(SEL)*aSelector* **to:**(id)*anObject* **forAllCells:**(BOOL)*flag*

Iterates through all of the cells in the NSMatrix (if *flag* is YES), or just the selected cells in the NSMatrix (if *flag* is NO), sending *aSelector* to *anObject* for each. Iteration begins with the cell in the upper-left corner of the NSMatrix, proceeding through the appropriate entries in the first row, then on to the next.

aSelector must represent a method that takes a single argument: the **id** of the current cell in the iteration. *aSelector*'s return value must be a BOOL. If *aSelector* returns NO for any cell, **sendAction:to:forAllCells:**

terminates immediately, without sending the message to the remaining cells. If it returns YES, **endAction: to:forAllCells:** proceeds on to the next cell.

This method is *not* invoked to send action messages to target objects in response to mouse-down events in the NSMatrix. Instead, you can invoke it if you want to have multiple cells in an NSMatrix interact with an object. For example you could use it to verify the titles in a list of items, or to enable a series of radio buttons based on their purpose in relation to *anObject*.

sendDoubleAction

– (void)**sendDoubleAction**

If the NSMatrix has a double-click action, **sendDoubleAction** sends that message to the target of the NSMatrix. If not, then if the selected cell (as returned by **selectedCell**) has an action, that message is sent to the selected cell's target. Finally, if the selected cell also has no action, then the single-click action of the NSMatrix is sent to the target of the NSMatrix.

If the selected cell is disabled, this method does nothing.

Your code shouldn't invoke this method; it's sent in response to a double-click event in the NSMatrix. Override it if you need to change the search order for an action to send.

See also: – **sendAction**, – **ignoresMultiClick** (NSControl)

setAllowsEmptySelection:

– (void)**setAllowsEmptySelection:**(BOOL)*flag*

If *flag* is YES, then the NSMatrix will allow one or zero cells to be selected. If *flag* is NO, then the NSMatrix will allow one and only one cell (not zero cells) to be selected. This setting has effect only in the NSRadioModeMatrix selection mode.

setAutoscroll:

– (void)**setAutoscroll:**(BOOL)*flag*

If *flag* is YES and the NSMatrix is in a scrolling view, it will be automatically scrolled whenever a the mouse is dragged outside the NSMatrix after a mouse-down event within the bounds of the NSMatrix.

setAutosizesCells:

– (void)**setAutosizesCells:**(BOOL)*flag*

If *flag* is YES, then whenever the NSMatrix is resized, the sizes of the cells change in proportion, keeping the inter-cell space constant; further, this method verifies that the cell sizes and inter-cell spacing add up to the exact size of the NSMatrix, adjusting the size of the cells and updating the NSMatrix if they don't. If *flag* is NO, then the inter-cell space changes when the NSMatrix is resized, with the cell size remaining constant.

setBackgroundColor:

– (void)**setBackgroundColor:**(NSColor *)*aColor*

Sets the background color for the NSMatrix to *aColor*, and redraws the NSMatrix. This color is used to fill the space between cells or the space behind any non-opaque cells. The default background color is NSColor's **controlColor**.

See also: – **drawsBackground**, – **setCellBackgroundColor:**

setCellBackgroundColor:

– (void)**setCellBackgroundColor:**(NSColor *)*aColor*

Sets the background color for the cells in the NSMatrix to *aColor*. This color is used to fill the space behind non-opaque cells. The default cell background color is NSColor's **controlColor**.

See also: – **drawsCellBackground**, – **setBackgroundColor:**

setCellClass:

– (void)**setCellClass:**(Class)*factoryId*

Configures the receiver to use instances of *factoryId* when creating new cells. *factoryId* should be the **id** of a subclass of NSCell, which can be obtained by sending the **class** message to either the NSCell subclass object or to an instance of that subclass. The default cell class is that set with the class method **setCellClass:** , or NSActionCell if no other default cell class has been specified.

You only need to use this method with matrices initialized with **initWithFrame:**, since the other initializers allow you to specify an instance-specific cell class or cell prototype.

See also: – **addColumn**, – **addRow**, – **insertColumn:**, – **insertRow:**, – **makeCellAtRow:column:**, – **setPrototype:**

setCellSize:

– (void)**setCellSize:**(NSSize)*aSize*

Sets the width and the height of each of the cells in the NSMatrix to those in *aSize*. This may change the size of the NSMatrix. Does not redraw the NSMatrix.

See also: – **calcSize** (NSControl)

setDelegate:

– (void)**setDelegate:**(id)*anObject*

Sets the delegate for messages from the field editor.

See also: – **textShouldBeginEditing:**, – **textShouldEndEditing:**

setDoubleAction:

– (void)**setDoubleAction:**(SEL)*aSelector*

Makes *aSelector* the action sent to the target of the NSMatrix when the user double-clicks a cell. A double-click action is always sent after the appropriate single-click action, which is the cell's single-click action, if it has one, or the NSMatrix's single-click action., otherwise. If *aSelector* is a valid selector, this method also sets the ignoresMultiClick flag to NO; otherwise, it leaves the flag unchanged.

If an NSMatrix has no double-click action set, then by default a double-click is treated as a single-click.

See also: – **sendDoubleAction**, – **setAction:** (NSControl), – **setTarget:** (NSControl)

setDrawsBackground:

– (void)**setDrawsBackground:**(BOOL)*flag*

Sets whether the receiver draws its background (the space between the cells).

See also: – **backgroundColor**, – **setDrawsCellBackground:**

setDrawsCellBackground:

– (void)**setDrawsCellBackground:**(BOOL)*flag*

Sets whether the receiver draws the background within each of its cells.

See also: – **cellBackgroundColor**, – **setDrawsBackground:**

setErrorAction:

– (void)**setErrorAction:**(SEL)*aSelector*

Sets the action that's sent to the target of the NSMatrix when the user enters an illegal value for the selected cell.

See also: – **action** (NSControl), – **target** (NSControl)

setIntercellSpacing:

– (void)**setIntercellSpacing:**(NSSize)*aSize*

Sets the vertical and horizontal spacing between cells in the NSMatrix. By default, both values are 1.0 in the NSMatrix's coordinate system.

See also: – **cellSize**

setKeyCell:

– (void)**setKeyCell:**(NSCell *)*aCell*

Sets to *aCell* the cell that will be clicked when the user presses the Return key.

See also: – **setNextText:**, – **setTabKeyTraversesCells:**

setMode:

– (void)**setMode:**(NSMatrixMode)*aMode*

Sets the selection mode of the NSMatrix. Possible values for *aMode* are defined in NSMatrix.h, and include:

Tracking mode	Description
NSTrackModeMatrix	Cells track the mouse, but do not highlight
NSHighlightModeMatrix	Cells highlight as they track the mouse
NSRadioModeMatrix	Allows no more than one selected cell
NSListModeMatrix	Cells are highlighted, but don't track the mouse

These modes are explained in detail in the class description.

See also: – **initWithFrame:mode:cellClass:numberOfRows:numberOfColumns:**, – **initWithFrame:mode:prototype:numberOfRows:numberOfColumns:**

setNextText:

– (void)**setNextText:**(id)*anObject*

If the NSMatrix doesn't already have a next key view, inserts *anObject* after the receiver in the key view loop of the receiver's NSWindow. *anObject* thus becomes the object that would be selected if the user presses Tab while editing the last text cell in the NSMatrix. If the NSMatrix already has a next key view, this method does nothing.

See also: – **setNextKeyView:** (NSView), – **setTabKeyTraversesCells:**

setPreviousText:

– (void)**setPreviousText:**(id)*anObject*

If *anObject* doesn't already have a next key view, inserts the receiver after *anObject* in the key view loop of *anObject*'s NSWindow. If *anObject* already has a next key view, this method does nothing.

See also: – **setNextKeyView:** (NSView)

setPrototype:

– (void)**setPrototype:**(NSCell *)*aCell*

Sets the prototype cell that's copied whenever a new cell needs to be created.

See also: – **initWithFrame:mode:prototype:numberOfRows:numberOfColumns:**,
– **makeCellAtRow:column:**

setScrollable:

– (void)**setScrollable:**(BOOL)*flag*

If *flag* is YES, sets all the cells to be scrollable, so that the text they contain scrolls to remain in view if the user types past the edge of the cell. If *flag* is NO, all cells are made to be non-scrolling. The prototype cell, if there is one, is also set accordingly.

See also: – **prototype**, – **setScrollable:** (NSCell)

setSelectionByRect:

– (void)**setSelectionByRect:**(BOOL)*flag*

Sets whether the user can select a rectangle of cells in the NSMatrix by dragging the cursor. If *flag* is NO, selection is on a row-by-row basis. The default is YES.

See also: – **setSelectionFrom:to:anchor:highlight:**

setSelectionFrom:to:anchor:highlight:

– (void)**setSelectionFrom:**(int)*startPos* **to:**(int)*endPos* **anchor:**(int)*anchorPos* **highlight:**(BOOL)*lit*

Programmatically selects a range of cells. *startPos*, *endPos*, and *anchorPos* are cell positions, counting from 0 at the upper left cell of the NSMatrix, in row order. For example, the third cell in the top row would be number 2.

startPos and *endPos* are used to mark where the user would have pressed the mouse button and released it, respectively. *anchorPos* locates the "last selected cell" with regard to extending the selection by Shift- or Alternate-clicking. Finally, *lit* determines whether cells selected by this method should be highlighted.

See also: – **isSelectionByRect**, – **selectedCells**

setState:atRow:column:

– (void)**setState:**(int)*value* **atRow:**(int)*row* **column:**(int)*column*

Sets the state of the cell at *row* and *column* to *value*. For radio-mode matrices, if *value* is non-zero the specified cell is selected before its state is set to *value*. If *value* is zero and the receiver is a radio-mode matrix, then the currently-selected cell is deselected (providing that empty selection is allowed).

This method redraws the affected cell.

See also: – **allowsEmptySelection**, – **setState:** (NSCell), – **selectCellAtRow:column:**

setTabKeyTraversesCells:

– (void)**setTabKeyTraversesCells:**(BOOL)*flag*

Sets whether pressing the Tab key advances the key cell to the next selectable cell in the NSMatrix. If *flag* is NO, or if there aren't any selectable cells after the current one, when the user presses the Tab key the next view in the window becomes key. Pressing Shift-Tab causes the key cell to advance in the opposite direction (if *flag* is NO, or if there aren't any selectable cells before the current one, the previous view in the window becomes key).

See also: – **selectKeyViewFollowingView:** (NSWindow), – **selectNextKeyView:** (NSWindow), – **setKeyCell:**, – **setNextText:**

setValidateSize:

– (void)**setValidateSize:**(BOOL)*flag*

If *flag* is YES, then the size information in the NSMatrix is assumed to be correct. If *flag* is NO, then **calcSize** will be invoked before any further drawing is done.

See also: – **calcSize** (NSControl)

sizeToCells

– (void)**sizeToCells**

Changes the width and the height of the NSMatrix frame so that it exactly contains the cells. Does not redraw the NSMatrix.

See also: – **setFrameSize:** (NSView), – **sizeToFit** (NSControl)

sortUsingFunction:context:

– (void)**sortUsingFunction:**(int (*)(id, id ,void *))*comparator* **context:**(void *)*context*

Sorts the receiver's cells in ascending order as defined by the comparison function *comparator*. The comparison function is used to compare two elements at a time and should return NSOrderedAscending if the first element is smaller than the second, NSOrderedDescending if the first element is larger than the second, and NSOrderedSame if the elements are equal. Each time the comparison function is called, it's passed *context* as its third argument. This allows the comparison to be based on some outside parameter, such as whether character sorting is case-sensitive or case-insensitive.

See also: – **sortUsingFunction:context:** (NSMutableArray)

sortUsingSelector:

– (void)**sortUsingSelector:**(SEL)*comparator*

Sorts the receiver's cells in ascending order as defined by the comparison method *comparator*. The *comparator* message is sent to each object in the matrix, and has as its single argument another object in the array. The comparison method is used to compare two elements at a time and should return NSOrderedAscending if the receiver is smaller than the argument, NSOrderedDescending if the receiver is larger than the argument, and NSOrderedSame if they are equal.

See also: – **sortUsingSelector:** (NSMutableArray)

tabKeyTraversesCells

– (BOOL)**tabKeyTraversesCells**

Returns whether pressing the Tab key advances the key cell to the next selectable cell in the NSMatrix.

See also: – **keyCell**, – **setTabKeyTraversesCells:**

textDidBeginEditing:

– (void)**textDidBeginEditing:**(NSNotification *)*notification*

Invoked when there's a change in the text after the receiver gains first responder status. This method's default behavior is to post an NSControlTextDidBeginEditingNotification along with the receiving object to the default notification center. The posted notification's user info contains the contents of *notification*'s user info dictionary, plus an additional key/value pair. The additional key is "NSFieldEditor"; the value for this key is the text object that began editing.

See also: – **textDidChange:**, – **textDidEndEditing:**, – **textShouldBeginEditing:**

textDidChange:

– (void)**textDidChange:**(NSNotification *)*notification*

Invoked upon a key-down event or paste operation that changes the receiver's contents. This method's default behavior is to pass this message on to the selected cell (if the selected cell responds to **textDidChange:**), and then to post an NSControlTextDidChangeNotification along with the receiving object to the default notification center. The posted notification's user info contains the contents of *notification*'s user info dictionary, plus an additional key/value pair. The additional key is "NSFieldEditor"; the value for this key is the text object that changed.

See also: – **textDidBeginEditing:**, – **textDidEndEditing:**

textDidEndEditing:

– (void)**textDidEndEditing:**(NSNotification *)*notification*

Invoked when text editing ends. This method's default behavior is to post an NSControlTextDidEndEditingNotification along with the receiving object to the default notification center. The posted notification's user info contains the contents of *notification*'s user info dictionary, plus an additional key/value pair. The additional key is "NSFieldEditor"; the value for this key is the text object that began editing. After posting the notification, **textDidEndEditing:** sends an **endEditing:** message to the selected cell, draws and makes the selected cell key, and then takes the appropriate action based on which key was used to end editing (Return, Tab, or Back-Tab).

See also: – **textDidBeginEditing:**, – **textDidChange:**, – **textShouldEndEditing:**

textShouldBeginEditing:

– (BOOL)**textShouldBeginEditing:**(NSText *)*textObject*

Invoked to let the NSTextField respond to impending changes to its text. This method's default behavior is to send **control:textShouldBeginEditing:** to the receiver's delegate (passing the receiver and *textObject* as parameters). **textShouldBeginEditing:** returns the value obtained from **control:textShouldBeginEditing:** , unless the delegate doesn't respond to that method, in which case it returns YES, thereby allowing text editing to proceed.

See also: – **delegate**

textShouldEndEditing:

– (BOOL)**textShouldEndEditing:**(NSText *)*textObject*

Invoked to let the NSTextField respond to impending loss of first-responder status. This method's default behavior checks the text field for validity; providing that the field contents are deemed valid, and providing that the delegate responds, **control:textShouldEndEditing:** is sent to the receiver's delegate (passing the receiver and *textObject* as parameters). If the contents of the text field aren't valid, **textShouldEndEditing:** sends the error action to the selected cell's target.

textShouldEndEditing: returns NO if the text field contains invalid contents, otherwise it returns the value passed back from **control:textShouldEndEditing:**.

See also: – **delegate**, – **errorAction**

NSMenu

Inherits From:	NSObject
Conforms To:	NSCoding NSCopying NSObject (NSObject)
Declared In:	AppKit/NSMenu.h

Class Description

This class defines an object that manages an application's menus. An NSMenu object displays a list of items that a user can choose from. When an item is clicked, the NSMenu object may either issue a command directly (by sending an action message to a target object) or bring up another menu (a *submenu*) that offers further choices. An NSMenu object's choices are implemented with NSMenuItem objects. Each menu item can be configured either to send its action message to a target or to open a submenu.

It's typically more convenient to use Interface Builder to construct your application's menus—see Interface Builder's Help for more information about using this application. NSMenu and NSMenuItem provide you with additional flexibility to contruct or modify your application's menus dynamically at run time.

Exactly one NSMenu created by the application is designated as the main menu for the application (with NSApplication's **setMainMenu:** method). Depending on the user interface of the host system, the main menu displays itself as a free-standing window with a title bar and a list of menu items, as a menu bar with no title, or in some other form. The form a menu takes in the user interface may limit which methods in the class's interface actually have an effect; for example, on Microsoft Windows submenus can't be detached. In all cases, however, methods return values that reflect the NSMenu object's actual state or ability (**isTornOff** always returns NO on Microsoft Windows, for example).

On Mach, a free-standing main menu is displayed on top of all other windows whenever the application is active; the user can move it by dragging its title bar. When a submenu is opened, it appears attached to the right of its supermenu with a title bar, allowing the user to drag it away from its supermenu so that it remains on the screen. A detached submenu displays a close button to allow the user to dismiss it (the main menu, of course, never displays a close button). If the user moves a menu window while a submenu is attached, the submenu follows its supermenu. If a menu window lies partly off-screen, when the user tracks the mouse pointer to the edge of the screen, by holding down the mouse button and dragging the mouse pointer, the menu temporarily shifts onto the screen (along with any attached super- or submenus), allowing the user to access all of its items.

Where the main menu appears as a menu bar (Windows, for example), the menu doesn't display a title, nor do its submenus. The submenus of a menu bar are typical drop-down menus, and submenus of these appear to the right or left, depending on the available screen space.

NSMenu supports the assignment of keyboard equivalents (command-key accelerators) to its menu items. On Microsoft Windows, the class also supports the assignment of mnemonics to menu items. Any menu item, except those that open submenus, can have a key equivalent, but whether they should have one depends on the user interface guidelines of the host system's. Unlike keyboard equivalents, mnemonics only function when their menu is active, and they can be assigned to menu items which open submenus.

The items that appear on menus belong to the NSMenuItem class, which simply adopts the NSMenuItem protocol and adds little other behavior. See the specifications of the NSMenuItem and NSMenuValidation protocols for more information.

Menu Autoenabling

By default, menus are *autoenabled*. "Autoenabling" refers to the ability of a menu to enable or disable its items after a user event by querying other objects in an application for the appropriate state. (A disabled menu item has a gray title and does not respond to mouse clicks or key equivalents.) For instance, if the user selects some text in a scroll view, the object responsible for managing that text could receive—as the target of menu items such as Cut, Copy, and Paste—the message **validateMenuItem:** for each of those items (see informal protocol NSMenuValidation). It would implement this message to evaluate the current context and return whether the menu item should be enabled.

An NSMenu object locates the "validator" object for a menu item by testing for the existence of the following objects, in the given order:

- The target of the menu item
- Any other object implementing the method corresponding to the menu item's action (that is, the search for an implementor proceeds up the *responder chain*; see the NSResponder specification for details)

By following these steps, the NSMenu ensures that the object receiving the action message is asked to validate the menu state.

If the NSMenu object cannot locate a "validator," it disables the menu item. If the validator responds to **validateMenuItem:**, NSMenu asks it for the enabled state. If the validator does not respond to the message, NSMenu enables the menu item.

You can turn off autoenabling by sending **setAutoenablesItems:** to the NSMenu object with an argument of NO. You should do this if your application explicitly controls the state of each of its menu items (see the NSMenuItem protocol method **setEnabled:**). Although autoenabling occurs automatically upon each user event, you can request it for other purposes with the **update** method.

Method Types

Controlling allocation zones	+ menuZone + setMenuZone:

Creating an NSMenu

– initWithTitle:

Setting up menu commands

– insertItem:atIndex:
– insertItemWithTitle:action:keyEquivalent:atIndex:
– addItem:
– addItemWithTitle:action:keyEquivalent:
– removeItem:
– removeItemAtIndex:
– itemChanged:

Finding menu items

– itemWithTag:
– itemWithTitle:
– itemAtIndex:
– numberOfItems
– itemArray

Finding indices of menu items

– indexOfItem:
– indexOfItemWithTitle:
– indexOfItemWithTag:
– indexOfItemWithTarget:andAction:
– indexOfItemWithRepresentedObject:
– indexOfItemWithSubmenu:

Managing submenus

– setSubmenu:forItem:
– submenuAction:
– attachedMenu
– isAttached
– isTornOff
– locationForSubmenu:
– supermenu
– setSupermenu:

Enabling and disabling menu items

– autoenablesItems
– setAutoenablesItems:
– update

Handling keyboard equivalents

– performKeyEquivalent:

Simulating mouse clicks

– performActionForItemAtIndex:

Setting the title

– setTitle:
– title

Setting the representing object

– setMenuRepresentation:
– menuRepresentation

Updating menu layout

– menuChangedMessagesEnabled
– setMenuChangedMessagesEnabled:
– sizeToFit

Displaying context-sensitive help

– helpRequested:

Class Methods

menuZone

+ (NSZone *)**menuZone**

Returns the zone from which NSMenus should be allocated, creating it if necessary.

setMenuZone:

+ (void)**setMenuZone:**(NSZone *)*zone*

Sets the zone from which NSMenus should be allocated to *zone*.

Instance Methods

addItem:

– (void)**addItem:**(id <NSMenuItem>)*newItem*

Adds the menu item *newItem* to the end of the receiving NSMenu. In its implementation this method invokes **insertItem:atIndex:**. Thus, the menu does not accept the menu item if it already belongs to another menu. After adding the menu item, the menu updates itself.

addItemWithTitle:action:keyEquivalent:

– (id <NSMenuItem>)**addItemWithTitle:**(NSString *)*aString*
action:(SEL)*aSelector*
keyEquivalent:(NSString *)*keyEquiv*

Adds a new item with title *aString*, action *aSelector*, and key equivalent *keyEquiv* to the end of the menu. Returns the new menu item. If you do not want the menu item to have a key equivalent, *keyEquiv* should be an empty string (@"") and not **nil**.

attachedMenu

– (NSMenu *)**attachedMenu**

Returns the menu currently attached to the receiver or **nil** if there's no such object.

autoenablesItems

– **(**BOOL)**autoenablesItems**

Returns whether the receiver automatically enables and disables its menu items based on the NSMenuValidation informal protocol. By default NSMenus do autoenable their menu items. See that protocol specification for more information.

See also: **– setAutoenablesItems:**

helpRequested:

– (void)**helpRequested:**(NSEvent *)*event*

Overridden by subclasses to implement specialized context-sensitive help behavior by causing the Help manager to display the help associated with the receiver. Never invoke this method directly.

See also: – **showContextHelpForObject:locationHint:** (NSHelpManager)

indexOfItem:

– (int)**indexOfItem:**(id <NSMenuItem>)*anObject*

Returns the index identifying the location of menu item *anObject* in the receiver. If no such menu item is in the menu, the method returns -1.

See also: – **insertItem:atIndex:**, – **itemAtIndex:**

indexOfItemWithRepresentedObject:

– (int)**indexOfItemWithRepresentedObject:**(id)*anObject*

Returns the index of the first menu item in the receiver that has *anObject* as its represented object. If no such menu item is in the menu, the method returns -1.

See also: – **insertItem:atIndex:**, – **itemAtIndex:**

indexOfItemWithSubmenu:

– (int)**indexOfItemWithSubmenu:**(NSMenu *)*anObject*

Returns the index of the menu item in the receiver that has submenu *anObject*. If no such menu item is in the menu, the method returns -1.

See also: – **insertItem:atIndex:**, – **itemAtIndex:**

indexOfItemWithTag:

– (int)**indexOfItemWithTag:**(int)*aTag*

Returns the index of the first menu item in the receiver identified by tag *aTag*. If no such menu item is in the menu, the method returns -1.

See also: – **insertItem:atIndex:**, – **itemAtIndex:**

indexOfItemWithTarget:andAction:

– (int)**indexOfItemWithTarget:**(id)*anObject* **andAction:**(SEL)*actionSelector*

Returns the index of the first menu item in the receiver that has a target of *anObject* and an action of *actionSelector*. If *actionSelector* is NULL, the first menu item in the receiver that has a target of *anObject* is returned. If no menu item matching these criteria is in the menu, the method returns -1.

See also: – **insertItem:atIndex:**, – **itemAtIndex:**

indexOfItemWithTitle:

– (int)**indexOfItemWithTitle:**(NSString *)*aTitle*

Returns the index of the first menu item in the receiver that has the title *aTitle*. If no such menu item is in the menu, the method returns -1.

See also: – **insertItem:atIndex:**, – **itemAtIndex:**

initWithTitle:

– (id)**initWithTitle:**(NSString *)*aTitle*

Initializes and returns a new menu using *aTitle* for its title and with autoenabling of menu items turned on. This method is the designated initializer for the class. Returns **self**.

See also: – **setAutoenablesItems:**

insertItem:atIndex:

– (void)**insertItem:**(id <NSMenuItem>)*newItem* **atIndex:**(int)*index*

Inserts the menu item *newItem* in the receiving NSMenu at location *index*. If the menu item already exists in another menu, it is not inserted. The method informs the object implementing the platform-specific look and behavior of the menu (the "menu representation") that the item has been inserted. It also causes the menu to update itself. This is a primitive method; if you create a subclass of NSMenu, this method must be overriden.

See also: – **addItem:**, – **itemAtIndex:**, **– removeItem:**

insertItemWithTitle:action:keyEquivalent:atIndex:

– (id <NSMenuItem>)**insertItemWithTitle:**(NSString *)*aString*
action:(SEL)*aSelector*
keyEquivalent:(NSString *)*keyEquiv*
atIndex:(unsigned int)*index*

Adds a new item at the location *index* in the menu that has the title *aString*, action *aSelector*, and key equivalent *keyEquiv*. Returns the new menu item. If you do not want the menu item to have a key equivalent, *keyEquiv* should be an empty string (@"") and not **nil**.

isAttached

– (BOOL)**isAttached**

Returns YES if the receiver is currently attached to another menu, NO otherwise. This method always returns NO on Microsoft Windows.

isTornOff

– (BOOL)**isTornOff**

Returns NO if the receiver is off-screen or attached to another menu (or if it's the main menu), YES otherwise. This method always returns NO on Microsoft Windows.

itemArray

– (NSArray *)**itemArray**

Returns the receiver's menu items.

See also: – **numberOfItems**

itemAtIndex:

– (id <NSMenuItem>)**itemAtIndex:**(int)*index*

Returns the menu item at location *index* of the receiver. It raises an exception if *index* is out of bounds.

itemChanged:

– (void)**itemChanged:**(id <NSMenuItem>)*anObject*

Invoked when menu item *anObject* is modified (for example, its title changes). The default implementation informs the "menu representation" object and causes the menu to update itself.

itemWithTag:

– (id <NSMenuItem>)**itemWithTag:**(int)*aTag*

Returns the first menu item in the receiver that has *aTag* as its tag.

itemWithTitle:

– (id <NSMenuItem>)**itemWithTitle:**(NSString *)*aString*

Returns the first menu item in the receiver that has *aString* as its title.

locationForSubmenu:

– (NSPoint)**locationForSubmenu:**(NSMenu *)*aSubmenu*

On Mach, returns the screen coordinates where *aSubmenu* will be displayed when it's opened as a submenu of the receiver (regardless of its current location). On Microsoft Windows, the coordinates that are reutrned are not meaningful.

menuChangedMessagesEnabled

– (BOOL)**menuChangedMessagesEnabled**

Returns YES if messages are being sent to the application's windows upon each change to the menu, NO otherwise.

See also: – **setMenuChangedMessagesEnabled:**

menuRepresentation

– (id)**menuRepresentation**

Returns the object that implements the "look and feel" of the menu for a particular platform. For Macintosh and Mach platforms, this object is NSMenuView.

See also: – **setMenuRepresentation:**

numberOfItems

– (int)**numberOfItems**

Returns the number of menu items in the receiver, including separator items.

See also: – **itemArray**

performActionForItemAtIndex:

– (void)**performActionForItemAtIndex:**(int)*index*

Causes the application to send the action message of the menu item at *index* to it's target. If a target is not specified, the message is sent to the first responder. As a side effect, this method posts NSMenuWillSendActionNotification and NSMenuDidSendActionNotification.

performKeyEquivalent:

– (BOOL)**performKeyEquivalent:**(NSEvent *)*theEvent*

Searches for a menu item in the receiver, or on Microsoft Windows in any of its submenus as well, whose key equivalent exactly matches the character, or character sequence, of the keyboard event *theEvent* and whose modifier flags match the key-equivalent modifier mask in *theEvent*, and causes that item to send its action message.

removeItem:

– (void)**removeItem:**(id <NSMenuItem>)*anItem*

Removes *anItem* from the receiver.

removeItemAtIndex:

– (void)**removeItemAtIndex:**(int)*index*

Removes the menu item at location *index*.

setAutoenablesItems:

– (void)**setAutoenablesItems:**(BOOL)*flag*

Controls whether the receiver automatically enables and disables its menu items based on delegates implementing the NSMenuValidation informal protocol. If *flag* is YES, menu items are automatically enabled and disabled. If *flag* is NO, menu items are not automatically enabled or disabled. See the NSMenuValidation protocol specification for more information.

See also: **– autoenablesItems**

setMenuChangedMessagesEnabled:

– (void)**setMenuChangedMessagesEnabled:**(BOOL)*flag*

Controls whether the receiver sends messages to the application's windows upon each menu change. To avoid the "flickering" effect of many successive menu changes, invoke this method with NO as *flag*, make changes to the menu, and invoke the method again with YES as *flag*. This has the effect of batching changes and having them applied all at once.

See also: – **menuChangedMessagesEnabled**

setMenuRepresentation:

– (void)**setMenuRepresentation:**(id)*menuRep*

Sets the object that implements the "look and feel" of the menu for a particular platform. For Macintosh and Mach platforms, this object is NSMenuView. On any supported platform except Windows, you can set your own NSMenuView subclass as the menu representation.

See also: – **menuRepresentation**

setSubmenu:forItem:

– (void)**setSubmenu:**(NSMenu *)*aMenu* **forItem:**(id <NSMenuItem>)*anItem*

Makes *aMenu* a submenu controlled by *anItem*, automatically setting *anItem*'s action to **submenuAction:**.

setSupermenu:

– (void)**setSupermenu:**(NSMenu *)*supermenu*

Sets the receiver's supermenu (which obviously must be a submenu) to *supermenu*. You should never invoke this method directly, although you may override it.

See also: **– supermenu**

setTitle:

– (void)**setTitle:**(NSString *)*aString*

Sets the receiver's title to *aString*.

See also: **– title**

sizeToFit

– (void)**sizeToFit**

Resizes the receiver to exactly fit its items. On Microsoft Windows, this method has no effect.

submenuAction:

– (void)**submenuAction:**(id)*sender*

This is the action method assigned to menu items that open submenus. Never invoke this method directly.

supermenu

– (NSMenu *)**supermenu**

Returns the receiver's supermenu or **nil** if it has none.

title

– (NSString *)**title**

Returns the receiver's title.

See also: **– setTitle:**

update

– (void)**update**

Enables or disables the receiver's menu items based on the NSMenuValidation informal protocol and sizes the menu to fit its current menu items if necessary. See the NSMenuValidation protocol specification for more information.

Notifications

NSMenuDidSendActionNotification

This notification contains a notification object and a userInfo dictionary. The notification object is the NSMenu containing the chosen menu item. The userInfo dictionary contains these keys and values:

Key	Value
@"MenuItem"	The menu item that was chosen.

This notification is posted just after the application invokes the action method (carried as instance data by the menu item) in the menu item's target object or, if no target is specified, in the first object in the responder chain that implements the action method.

NSMenuWillSendActionNotification

This notification contains a notification object and a userInfo dictionary. The notification object is the NSMenu containing the chosen menu item. The userInfo dictionary contains these keys and values:

Key	Value
@"MenuItem"	The menu item that was chosen.

This notification is posted just after the application invokes the action method (carried as instance data by the menu item) in the menu item's target object or, if no target is specified, in the first object in the responder chain that implements the action method.

NSMenuItem

Inherits From:	NSObject
Conforms To:	NSMenuItem NSObject (NSObject)
Declared In:	AppKit/NSMenuItem.h

Class Description

NSMenuItem is the class that OPENSTEP uses to implement the functionality of the NSMenuItem protocol. The NSMenuItem class defines objects that are used as command items in menus. In addition to implementing all of the methods in the NSMenuItem protocol, the NSMenuItem class also includes some private functionality that is needed to maintain binary compatibility with other components of OPENSTEP. Because of this, you cannot replace the NSMenuItem class with a different class which conforms to the NSMenuItem protocol. You may, however, subclass the NSMenuItem class if necessary.

The appearance of NSMenuItem objects is tailored to match the user interface of the host system, presently Macintosh, Mach, or Microsoft Windows.

See the NSMenu class specification and the NSMenuItem protocol specification for more information on menus.

Adopted Protocols

NSMenuItem

+ setUsesUserKeyEquivalents:
+ usesUserKeyEquivalents
- action
- hasSubmenu
- isEnabled
- keyEquivalent
- keyEquivalentModifierMask
- mnemonic
- mnemonicLocation
- representedObject
- setAction:
- setEnabled:
- setKeyEquivalent:
- setKeyEquivalentModifierMask:
- setMnemonicLocation:
- setRepresentedObject:
- setTag:
- setTarget:
- setTitle:
- setTitleWithMnemonic:
- tag
- target
- title
- userKeyEquivalent

NSMutableParagraphStyle

Inherits From: NSParagraphStyle : NSObject

Conforms To: NSCoding (NSParagraphStyle)
NSCopying (NSParagraphStyle)
NSMutableCopying (NSParagraphStyle)
NSObject (NSObject)

Declared In: AppKit/NSParagraphStyle.h

Class Description

NSMutableParagraphStyle adds methods to its superclass, NSParagraphStyle, for changing the values of the sub-attributes in a paragraph style attribute. See the NSParagraphStyle and NSAttributedString specifications for more information.

Method Types

Setting tab stops

– setTabStops:
– addTabStop:
– removeTabStop:

Setting other style information

– setParagraphStyle:
– setAlignment:
– setFirstLineHeadIndent:
– setHeadIndent:
– setTailIndent:
– setLineBreakMode:
– setMaximumLineHeight:
– setMinimumLineHeight:
– setLineSpacing:
– setParagraphSpacing:

Instance Methods

addTabStop:

– (void)**addTabStop:**(NSTextTab *)*tabStop*

Adds *tabStop* to the receiver.

See also: – **removeTabStop:**, – **setTabStops:**, – **tabStops** (NSParagraphStyle)

removeTabStop:

– (void)**removeTabStop:**(NSTextTab *)*tabStop*

Removes the first text tab whose location and type are equal to those of *tabStop*.

See also: – **addTabStop:**, – **setTabStops:**, – **tabStops** (NSParagraphStyle)

setAlignment:

– (void)**setAlignment:**(NSTextAlignment)*alignment*

Sets the alignment of the receiver to *alignment*, which may be one of:

NSLeftTextAlignment
NSRightTextAlignment
NSCenterTextAlignment
NSJustifiedTextAlignment
NSNaturalTextAlignment

See also: – **alignment** (NSParagraphStyle)

setFirstLineHeadIndent:

– (void)**setFirstLineHeadIndent:**(float)*aFloat*

Sets the distance in points from the leading margin of a text container to the beginning of the paragraph's first line to *aFloat*. This value must be nonnegative.

See also: – **setHeadIndent:**, – **setTailIndent:**, – **firstLineHeadIndent** (NSParagraphStyle)

setHeadIndent:

– (void)**setHeadIndent:**(float)*aFloat*

Sets the distance in points from the leading margin of a text container to the beginning of lines other than the first to *aFloat*. This value must be nonnegative.

See also: – **setFirstLineHeadIndent:**, – **setTailIndent:**, – **headIndent** (NSParagraphStyle)

setLineBreakMode:

– (void)**setLineBreakMode:**(NSLineBreakMode)*mode*

Sets the mode used to break lines in a layout container to *mode*, which may be one of:

NSLineBreakByWordWrapping
NSLineBreakByCharWrapping
NSLineBreakByClipping
NSLineBreakByTruncatingHead
NSLineBreakByTruncatingTail
NSLineBreakByTruncatingMiddle

See the description of **lineBreakMode** in the NSParagraphStyle class specification for descriptions of these values.

setLineSpacing:

– (void)**setLineSpacing:**(float)*aFloat*

Sets the space in points added between lines within the paragraph to *aFloat*. This value must be nonnegative.

See also: – **setMaximumLineHeight:**, – **setMinimumLineHeight:**, – **setParagraphSpacing:**, – **lineSpacing** (NSParagraphStyle)

setMaximumLineHeight:

– (void)**setMaximumLineHeight:**(float)*aFloat*

Sets the maximum height that any line in the paragraph style will occupy, regardless of the font size or size of any attached graphic, to *aFloat*. Glyphs and graphics exceeding this height will overlap neighboring lines; however, a maximum height of zero implies no line height limit. This value must be nonnegative.

Note: Although this limit applies to the line itself, line spacing adds extra space between adjacent lines.

See also: – **setMinimumLineHeight:**, – **setLineSpacing:**, – **maximumLineHeight** (NSParagraphStyle)

setMinimumLineHeight:

– (void)**setMinimumLineHeight:**(float)*aFloat*

Sets the minimum height that any line in the paragraph style will occupy, regardless of the font size or size of any attached graphic, to *aFloat*. This value must be nonnegative.

See also: – **setMaximumLineHeight:**, – **setLineSpacing:**, – **minimumLineHeight** (NSParagraphStyle)

setParagraphSpacing:

– (void)**setParagraphSpacing:**(float)*aFloat*

Sets the space added at the end of the paragraph to separate it from the following paragraph to *aFloat*. This value must be nonnegative.

See also: – **setLineSpacing:**, – **paragraphSpacing** (NSParagraphStyle)

setParagraphStyle:

– (void)**setParagraphStyle:**(NSParagraphStyle *)*aStyle*

Replaces the sub-attributes of the receiver with those in *aStyle*.

setTabStops:

– (void)**setTabStops:**(NSArray *)*tabStops*

Replaces the tab stops in the receiver with *tabStops*.

See also: – **addTabStop:**, – **removeTabStop:**, – **tabStops** (NSParagraphStyle)

setTailIndent:

– (void)**setTailIndent:**(float)*aFloat*

Sets the distance in points from the margin of a text container to the end of lines to *aFloat*. If positive, this is the distance from the leading margin (for example, the left margin in left-to-right text). If zero or negative, it's the distance from the trailing margin.

For example, to create a paragraph style that fits exactly in a 2-inch wide container, set its head indent to 0.0 and its tail indent to 0.0. To create a paragraph style with quarter-inch margins, set its head indent to 0.25 and its tail indent to –0.25.

See also: – **setHeadIndent:**, – **setFirstLineHeadIndent:**, – **tailIndent** (NSParagraphStyle)

NSOpenPanel

Inherits From: NSSavePanel : NSObject

Conforms To: NSObject (NSObject)

Declared In: AppKit/NSOpenPanel.h

Note: The inheritance and conformance information shown above applies only to NSOpenPanel on OpenStep for Windows. On Mach, NSOpenPanel inherits from (in this order) NSSavePanel, NSPanel, NSWindow, NSResponder, and NSObject.

Class Description

NSOpenPanel provides the Open dialog on OpenStep for Windows or the Open panel on the OpenStep for Mach user interface. Applications use the Open panel (or dialog) as a convenient way to query the user for the name of a file to open. The Open panel can only be run modally.

Most of this class's behavior is defined by its superclass, NSSavePanel. NSOpenPanel adds to this behavior by:

- Letting you specify the types (by file-name extension) of the items that will appear in the panel
- Letting the user select files, directories, or both
- Letting the user select multiple items at a time

Typically, you access an NSOpenPanel by invoking the **openPanel** method. When the class receives an **openPanel** message, it tries to reuse an existing panel rather than create a new one. If a panel is reused, its attributes are reset to the default values so that the effect is the same as receiving a new panel. Because Open dialogs and panels may be reused, you shouldn't modify the instance returned by **openPanel** except through the methods listed below (and through those inherited from NSSavePanel). For example, you can set the panel's title and whether it allows multiple selection, but not the arrangement of the buttons within the panel. If you must modify the Open panel substantially, create and manage your own instance using the **alloc...** and **init...** methods rather than the **openPanel** method.

The following code example shows the NSOpenPanel displaying only files with extensions of ".td" and allowing multiple selection. If the user makes a selection and clicks the OK button (that is, **runModalForDirectory:file:types:** returns NSOKButton), this method opens each selected file:

```
- (void)openDoc:(id)sender
{
    int result;
    NSArray *fileTypes = [NSArray arrayWithObject:@"td"];
    NSOpenPanel *oPanel = [NSOpenPanel openPanel];

    [oPanel setAllowsMultipleSelection:YES];
    result = [oPanel runModalForDirectory:NSHomeDirectory() file:nil
         types:fileTypes];
    if (result == NSOKButton) {
        NSArray *filesToOpen = [oPanel filenames];
        int i, count = [filesToOpen count];
        for (i=0; i<count; i++) {
            NSString *aFile = [filesToOpen objectAtIndex:i];
            id currentDoc = [[ToDoDoc alloc] initWithFile:aFile];
        }
    }
}
```

Method Types

Obtaining the shared instance

+ openPanel

Running the panel modally

– runModalForDirectory:file:types:
– runModalForTypes:

Getting the user selection

– filenames

Allowing browser selections

– setCanChooseFiles:
– canChooseFiles
– setCanChooseDirectories:
– canChooseDirectories

Allowing multiple selections

– setAllowsMultipleSelection:
– allowsMultipleSelection

Class Methods

openPanel

+ (NSOpenPanel *)**openPanel**

Returns a "recycled" NSOpenPanel or, if one doesn't yet exist, creates it before returning it. New and recycled NSOpenPanels are reset to default values, which include selection of single files only.

Instance Methods

allowsMultipleSelection

– (BOOL)**allowsMultipleSelection**

Returns whether the NSOpenPanel's browser allows the user to open multiple files (and directories) at a time. If multiple files or directories are allowed, then the **filename** method—inherited from NSSavePanel—returns a non-**nil** value only if one and only one file is selected. By contrast, NSOpenPanel's **filenames** method always returns the selected files, even if only one file is selected.

See also: – **filename**(NSSavePanel), – **filenames**, – **setAllowsMultipleSelection:**

canChooseDirectories

– (BOOL)**canChooseDirectories**

Returns whether the Open dialog or panel allows the user to choose directories to open.

See also: – **setCanChooseDirectories:**

canChooseFiles

– (BOOL)**canChooseFiles**

Returns whether the Open dialog or panel allows the user to choose files to open.

See also: – **setCanChooseFiles:**

filenames

– (NSArray *)**filenames**

Returns an array containing the absolute paths (as NSString objects) of the selected files and directories. If multiple selections aren't allowed, the array contains a single name. The **filenames** method is preferable over NSSavePanel's **filename** to get the name or names of files and directories that the user has selected.

runModalForDirectory:file:types:

– (int)**runModalForDirectory:**(NSString *)*directory*
file:(NSString *)*filename*
types:(NSArray *)*fileTypes*

Displays the NSOpenPanel and begins a modal event loop that is terminated when the user clicks either OK or Cancel, resulting in the return of NSOKButton or NSCancelButton, respectively. The NSOpenPanel displays the files in *directory* (an absolute directory path) that match the types in *fileTypes* (an NSArray of file extensions). If *directory* is **nil** the default directory on Mach is the application directory; on Windows the default directory is the root directory of the drive on which the application resides. If all files in a directory should appear in the browser, *fileTypes* should be **nil**. You can control whether directories and files appear in the browser with the **setCanChooseDirectories:** and **setCanChooseFiles:** methods. The *filename* argument specifies a particular file in *startDir* that is selected when the Open dialog or panel is presented to the user; otherwise, *filename* should be **nil**.

See also: – **runModalForTypes:**

runModalForTypes:

– (int)**runModalForTypes:**(NSArray *)*fileTypes*

Invokes the **runModalForDirectory:file:types:** method, using **nil** for both file and directory (see description of **runModalForDirectory:file:types:**). The *fileTypes* argument is an NSArray containing the extensions of files to be shown in the browser. Returns NSOKButton (if the user clicks the OK button) or NSCancelButton (if the user clicks the Cancel button).

setAllowsMultipleSelection:

– (void)**setAllowsMultipleSelection:**(BOOL)*flag*

Sets whether the user can select multiple files (and directories) at one time for opening.

See also: – **allowsMultipleSelection**

setCanChooseDirectories:

– (void)**setCanChooseDirectories:**(BOOL)*flag*

Sets whether the user can select directories in the NSOpenPanel's browser. When a directory is selected, the OK button is enabled only if *flag* is YES.

See also: – **canChooseDirectories**

setCanChooseFiles:

– (void)**setCanChooseFiles:**(BOOL)*flag*

Sets whether the user can select files in the NSOpenPanel's browser or type the files to be accepted.

See also: – **canChooseFiles**

NSOutlineView

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

Class at a Glance

Purpose

An NSOutlineView object uses a row-and-column format to display hierarchical data that can be expanded and collapsed, such as directories and files in a file system. A user can expand and collapse rows, edit values, and resize and rearrange columns.

Principal Attributes

- Expands and collapses rows
- Works with NSTableView
- Gets data from object you provide
- Retrieves only data that needs to be displayed
- Uses a delegate

Creation

Interface Builder

– initWithFrame: Designated initializer.

Commonly Used Methods

– dataSource:	Returns the object that provides the data to be displayed (method of NSTableView).
– numberOfRows:	Returns the number of rows in the NSOutlineView (method of NSTableView).
– collapseItem:	Causes an item to be collapsed.
– expandItem:	Causes an item to be expanded.
– reloadItem:reloadChildren:	Informs the NSOutlineView that data for an item has changed and needs to be retrieved and redisplayed.

Class Description

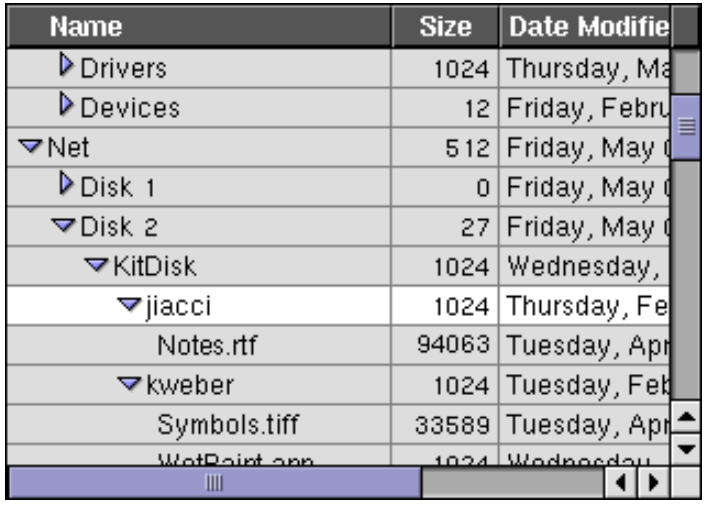
Note: The NSOutlineView class and its supporting informal protocol NSOutlineDataSource are under development and Interface Builder does not yet include support for working with it. If you want to use an NSOutlineView, you can instantiate it programmatically, or, in Interface Builder, you can drag a table view onto your interface and change its class name to NSOutlineView..

Before reading about NSOutlineView, you should read the documentation for its parent class, NSTableView. An NSOutlineView displays data for a set of related items, with rows representing individual items and columns representing the attributes of those items. As with an NSTableView, each item in an outline view represents a set of values for a particular real-world entity, such as an employee or a bank account. In addition, NSOutlineView provides the ability to expand or collapse rows containing hierarchical data.

An item in an NSOutlineView is *expandable* if it can contain other items. An expandable item is distinguished visually by an *expansion symbol* that varies according to the operating system. On the Macintosh, for example, an expandable item contains an *outline triangle*, which points to the right when the item is collapsed and points down when the item is expanded. Clicking on the outline triangle causes it to change position. It also causes the item to be expanded or collapsed, depending on the new state of the outline triangle. An item can be expanded even if it contains no items.

Items inside an expanded item are indented. By default, as a user expands or collapses nested items, the width of the column is resized so that it is just wide enough to display the widest item, based on the width of the items and their indentation in the hierarchy. Justification follows the current system justification. To turn off automatic resizing, use **setAutoResizesOutlineColumn:**. Note that an item may consist of text, an image, or anything else that can be drawn by a subclass of NSCell.

An NSOutlineView is typically displayed in an NSScrollView, as shown below.



In this illustration, the NSOutlineView consists of just the rows and columns that display values. The header is drawn by two auxiliary views: a header view that draws the column headers and a corner view that draws the blank square above the vertical scroller. These auxiliary views are described in the documentation for NSTableView.

Behavior Inherited from NSTableView

An outline view inherits much of its behavior from its parent class, NSTableView. As a result, many operations supported by a table view, such as selecting rows or columns, repositioning columns by dragging column headers, and so on, are also supported by an outline view. Your application has control of these features, and can configure the view's parameters to allow or disallow certain operations. For example, you might choose not to allow editing or rearranging for specific columns.

The NSTableView class also provides methods for working with data, responding to mouse clicks, setting grid attributes, editing cells, and performing other operations. For full information on these methods, see the documentation for NSTableView.

Data Source Messages

For more information on NSOutlineView's data source methods, see "Methods Implemented By the Data Source".

Delegate Messages

NSOutlineView adds several delegate messages to those defined by its superclass, NSTableView. In addition, it redefines certain NSTableView delegate methods to be item-based instead of row-based. Together, these methods give the delegate control over the appearance of individual cells in the table, over changes in selection, and over editing of cells.

Delegate methods that request permission to alert the selection or edit a value are invoked during user actions that affect the NSOutlineView, but are not invoked by programmatic changes to the view. When making changes programmatically, you decide whether you want the delegate to intervene and, if so, you send the appropriate message (checking first that the delegate responds to that message). Because the delegate methods involve the actual data displayed by the NSOutlineView, the delegate is typically the same object as the data source, though this is not a requirement.

NSOutlineView redefines these delegate messages based on similar messages in NSTableView:

> **outlineView:willDisplayCell:forTableColumn:item:** informs the delegate that the NSOutlineView is about to draw the cell specified by the passed column and item. The delegate can modify the NSCell provided to alter the display attributes for that cell; for example, making uneditable values display in italic or gray text.**outlineView:shouldSelectItem:** and **outlineView: shouldSelectTableColumn:** give the delegate control over whether the user can select a specified row or column (though the user can still reorder columns). This is useful for disabling a specified row or column. For example, in a database client application, when a user is editing a record you might want to not allow other users to select the same row.
>
> **selectionShouldChangeInOutlineView:** allows the delegate to deny a change in selection. For example, if the user is editing a cell and enters an improper value, the delegate can prevent the user from selecting or editing any other cells until a proper value has been entered into the original

cell.**outlineView:shouldEditTableColumn:item:** asks the delegate whether it's okay to edit the cell specified by the passed column and item. The delegate can approve or deny the request.

NSOutlineView defines these additional delegate messages:

outlineView:shouldExpandItem: asks the delegate whether it's okay to expand the specified item.

outlineView:willExpandItem: informs the delegate that the NSOutlineView is about to expand the specified item.

outlineView:shouldCollapseItem: asks the delegate whether it's okay to collapse the specified item.

outlineView:willCollapseItem: informs the delegate that the NSOutlineView is about to expand the specified item.

outlineView:willDisplayOutlineCell:forTableColumn:item: informs the delegate that the outline view is about to display the cell that includes the expansion symbol.

In addition to these methods, the delegate is automatically registered to receive messages corresponding to NSOutlineView notifications. These inform the delegate when the selection changes or is about to change, when a column is moved or resized, and when an item is expanded or collapsed:

Delegate Message	Notification
outlineViewColumnDidMove:	NSOutlineViewColumnDidMoveNotification
outlineViewColumnDidResize:	NSOutlineViewColumnDidResizeNotification
outlineViewSelectionDidChange:	NSOutlineViewSelectionDidChangeNotification
outlineViewSelectionIsChanging:	NSOutlineViewSelectionIsChangingNotification
outlineViewItemDidExpand:	NSOutlineViewItemDidExpandNotification
outlineViewItemDidCollapse:	NSOutlineViewItemDidCollapseNotification

Method Types

Creating an instance

– initWithFrame:

Expanding and collapsing the outline

– isExpandable:
– expandItem:
– expandItem:expandChildren:
– collapseItem:
– collapseItem:collapseChildren:
– isItemExpanded:

Redisplaying information

– reloadItem:
– reloadItem:reloadChildren:

Converting between items and rows

– itemAtRow:
– rowForItem:

Setting the outline column

– setOutlineTableColumn:
– outlineTableColumn

Setting the indentation

– levelForItem:
– levelForRow:
– setIndentationPerLevel:
– indentationPerLevel
– setIndentationMarkerFollowsCell:
– indentationMarkerFollowsCell

Persistence

autosaveExpandedItems
setAutosaveName:
setAutosaveExpandedItems:

Instance Methods

autoResizesOutlineColumn

– (BOOL) **autoResizesOutlineColumn**

Returns whether the outline view automatically resizes its outline column when the user exands or collapses items. The outline column contains the cells with the expansion symbols and is generally the first column. The default is YES (the outline column is automatically resized).

autosaveExpandedItems

– (BOOL) **autosaveExpandedItems**

Returns whether the expanded items in this outline view are automatically saved.

The outline view information is saved separately for each user and for each application that user uses. Note that if **autosaveName** returns **nil**, this setting is ignored and outline information isn't saved.

See also: – **autosaveName** (NSTableView), – **autosaveTableColumns** (NSTableView), **setAutosaveExpandedItems:**

collapseItem:

– (void)**collapseItem:**(id)*item*

Collapses *item* if *item* is expanded and expandable, otherwise does nothing. If collapsing takes place, posts item collapse notification.

See also: – **expandItem:**

collapseItem:collapseChildren:

– (void)**collapseItem:**(id)*item*
collapseChildren:(BOOL)*collapseChildren*

If *collapseChildren* is set to NO, collapses *item* only (identical to **collapseItem:**). If *collapseChildren* is set to YES, recursively collapses *item* and its children. For each item that is collapsed, posts item collapsed notification.

See also: – **collapseItem:**

expandItem:

– (void)**expandItem:**(id)*item*

Expands *item* if *item* is expandable and is not already expanded; otherwise, does nothing. If expanding takes place, posts item expanded notification.

See also: – **collapseItem:**

expandItem:expandChildren:

– (void)**expandItem:**(id)*item*
expandChildren:(BOOL)*expandChildren*

If *expandChildren* is set to NO, expands *item* only (identical to **expandItem:**). If *expandChildren* is set to YES, recursively expands *item* and its children. For each item that is expanded, posts item expanded notification.

See also: – **collapseItem:collapseChildren:**

indentationMarkerFollowsCell

– (BOOL)**indentationMarkerFollowsCell**

Returns YES if the expansion symbol in an expanded item is displayed in the cell with the item and NO if the symbol is displayed system-justified in the column. The default is YES.

See also: – **setIndentationMarkerFollowsCell:**

indentationPerLevel

– (float)**indentationPerLevel**

Returns current indentation per level, in points.

See also: – **setIndentationPerLevel:**

initWithFrame:

– (id)**initWithFrame:**(NSRect)*frameRect*

Initializes a newly allocated NSOutlineView with *frameRect* as its frame rectangle. This method is the designated initializer for the NSOutlineView class. It calls the **initWithFrame:** method of its superclass, NSTableView, then performs initialization specific to the outline view. It returns **self**.

The new NSOutlineView has a header view but has no columns; you can create NSTableColumn objects, set their titles and attributes, and add them to the new view with **addTableColumn:**. You must also set the NSOutlineView up in an NSScrollView with NSScrollView's **setDocView:** method. For more information, see the documentation for the NSTableView class.

It's usually more convenient to create an NSOutlineView using Interface Builder, which allows you to create an NSOutlineView already embedded in an NSScrollView, add and name the columns, and set up a data source.

isExpandable:

– (BOOL)**isExpandable:**(id)*item*

Returns YES if *item* is expandable—that is, *item* can contain other items.

See also: – **expandItem:**, – **isItemExpanded:**

isItemExpanded:

– (BOOL)**isItemExpanded**:(id)*item*

Returns YES if *item* is expanded.

See also: – **expandItem:**, – **isExpandable:**

itemAtRow:

– (id)**itemAtRow:**(int)*row*

Returns the item associated with *row*.

See also: – **rowForItem:**

levelForItem:

– (int)**levelForItem:**(id)*item*

Returns the indentation level for *item*.

See also: – **indentationPerLevel**, – **levelForRow:**

levelForRow:

– (int)**levelForRow:**(int)*row*

Returns the indentation level for *row*.

See also: – **indentationPerLevel**, – **levelForItem:**

outlineTableColumn

– (NSTableColumn *)**outlineTableColumn**

Returns the table column in which hierarchical data is displayed.

See also: – **setOutlineTableColumn:**

reloadItem:

– (void**)reloadItem:**(id)*item*

Reloads and redisplays the data for *item*.

See also: – **reloadItem:reloadChildren:**

reloadItem:reloadChildren:

– (void**)reloadItem:**(id)*item*
reloadChildren:(BOOL)*reloadChildren*

If *reloadChildren* is set to NO, reloads and redisplays the data for *item* only (identical to **reloadItem:**). If *reloadChildren* is set to YES, recursively reloads and redisplays the data for *item* and its children. It is not necessary, or efficient, to reload children if the item is not expanded.

See also: – **reloadItem:**

rowForItem:

– (int)**rowForItem:**(id)*item*

Returns the row associated with *item*.

See also: – itemAtRow**:**

setAutosaveName:

– (void)**setAutosaveName**:(NSString *)*name*

Sets the name under which outline view information is automatically saved to *name*. If *name* is different from the current name, this method also reads in the saved information and sets the outline view's options to match. As part of its implementation, this method calls NSTableView's **setAutosaveName:**.

The outline information is saved separately for each user and for each application that user uses.

Note that even when a outline view has an autosave name, it may not be saving outline information automatically. To set whether table information is being saved automatically, use **setAutosaveExpandedItems:** and **setAutosaveTableColumns:** (NSTableView).

See also: **autosaveName** (NSTableView), **setAutosaveTableColumns:** (NSTableView), **setAutosaveExpandedItems:**

setAutosaveExpandedItems:

– (void)**setAutosaveExpandedItems**:(BOOL)*flag*

Sets whether the expanded items in this outline view are automatically saved. If *flag* is different from the current value, this method also reads in the saved information and sets the outline view's options to match.

The outline information is saved separately for each user and for each application that user uses.

If **autosaveName** returns **nil** or if you haven't implemented the data source methods **outlineView: itemForPersistentObject:** and **outlineView:persistentObjectForItem:**, this setting is ignored and expanded item information isn't saved.

Note that you can have separate settings for **autosaveExpandedItems** and **autosaveTableColumns**, so you could, for example, save expanded item information, but not table column positions.

See also: **autosaveExpandedItems**, **setAutosaveName:**, **setAutosaveTableColumns:** (NSTableView)

- (void)setAutoResizesOutlineColumn: (BOOL)resize

– (void)**setAutoResizesOutlineColumn**:(BOOL)*resize*

Sets whether the outline view automatically resizes its outline column when the user exands or collapses an item. The outline column contains the cells with the expansion symbols and is generally the first column. The default is YES (the outline column is automatically resized).

setIndentationMarkerFollowsCell:

– (void)**setIndentationMarkerFollowsCell:**(BOOL)*drawInCell*

Sets whether the expansion symbol in an item should be displayed in the cell with the item or left-justified in the column. The default is YES (the symbol is displayed in the cell).

See also: – **indentationMarkerFollowsCell**

setIndentationPerLevel:

– (void)**setIndentationPerLevel:**(float)*newIndentLevel*

Sets indentation per level, in points, to *newIndentLevel*.

See also: – **indentationPerLevel**

setOutlineTableColumn:

– (void)**setOutlineTableColumn:** (NSTableColumn *)*outlineTableColumn*

Sets the table column in which hierarchical data is displayed to *outlineTableColumn*.

See also: – **outlineTableColumn**

Methods Implemented By the Data Source

outlineView:child:ofItem:

– (id)**outlineView:**(NSOutlineView *)*outlineView*
child:(int)*index*
ofItem:(id)*item*

Implemented by the data-source delegate. Children of a given parent *item* are accessed sequentially. This method should return the child item at the specified *index*. If *item* is **nil**, this method should return the appropriate child item of the root object.

See also: **– outlineView:numberOfChildrenOfItem:**

outlineView:isItemExpandable:

– (BOOL)**outlineView:**(NSOutlineView *)*outlineView* i**sItemExpandable:**(id)*item*

Implemented by the data-source delegate. This method should return true if *item* can be expanded to display its children.

outlineView:itemForPersistentObject:

– (id)**outlineView:**(NSOutlineView *)*outlineView* **itemForPersistentObject:**(id)*object*;

Implemented by the data-source delegate. Returns the item for the archived *object*. If the item is an archived object, this may return the object. You must implement this method if you are automatically saving expanded items (that is, **autoSaveExpandedItems** returns YES). When the outline view is restoring the

saved expanded items, this method is called for each expanded item, to translate the archived object to an outline view item.

See also:

outlineView:numberOfChildrenOfItem:

– (int)**outlineView:**(NSOutlineView *)*outlineView* **numberOfChildrenOfItem:**(id)*item*

Implemented by the data-source delegate. This method returns the number of child items encompassed by *item*. If *item* is **nil**, this method should return the number of children for the top-level item.

outlineView:objectValueForTableColumn:byItem:

– (id)**outlineView:**(NSOutlineView *)*outlineView*
objectValueForTableColumn:(NSTableColumn *)*tableColumn*
byItem:(id)*item*

Implemented by the data-source delegate. Returns the data object associated with the specified *item*. The *item* is located in the specified *tableColumn* of the view.

outlineView:persistentObjectForItem:

– (id)**outlineView:**(NSOutlineView *)*outlineView* **persistentObjectForItem:**(id)*item*;

Implemented by the data-source delegate. Returns an archived object for *item*. If the item is an archived object, this may return the item. You must implement this method if you are automatically saving expanded items (that is, **autoSaveExpandedItems** returns YES). When the outline view is saving the expanded items, this method is called for each expanded item, to translate the outline view item to an archived object.

outlineView:setObjectValue:forTableColumn:byItem:

– (void)**outlineView:**(NSOutlineView *)*outlineView*
setObjectValue:(id)*object*
forTableColumn:(NSTableColumn *)*tableColumn*
byItem:(id)*item*

Implemented by the data-source delegate. Sets the data object for the specified *item*. The *object* parameter contains the data to be set. The item is located in the specified *tableColumn* of the view.

Methods Implemented By the Delegate

Outline views support a data-source delegate in addition to the regular delegate object. The data-source delegate provides data and information about that data to the outline view. The regular delegate object handles all other delegate responsibilities for the outline view.

outlineViewColumnDidMove:

– (void)**outlineViewColumnDidMove:**(NSNotification *)*notification*

Invoked whenever the user moves a column in the outline view. This method is invoked as a result of posting an NSOutlineViewColumnDidMoveNotification.

See also: **NSOutlineViewColumnDidMoveNotification** (notification)

outlineViewColumnDidResize:

– (void)**outlineViewColumnDidResize:**(NSNotification *)*notification*

Invoked whenever the user resizes a column in the outline view. This method is invoked as a result of posting an NSOutlineViewColumnDidResizeNotification.

See also: **NSOutlineViewColumnDidResizeNotification** (notification)

outlineViewItemDidCollapse:

– (void)**outlineViewItemDidCollapse:**(NSNotification *)*notification*

Invoked whenever the user collapses an item in the outline view. This method is invoked as a result of posting an NSOutlineViewItemDidCollapseNotification.

See also: **NSOutlineViewItemDidCollapseNotification** (notification)

outlineViewItemDidExpand:

– (void)**outlineViewItemDidExpand:**(NSNotification *)*notification*

Invoked whenever the user expands an item in the outline view. This method is invoked as a result of posting an NSOutlineViewItemDidExpandNotification.

See also: **NSOutlineViewItemDidExpandNotification** (notification)

outlineViewItemWillCollapse:

– (void)**outlineViewItemWillCollapse:**(NSNotification *)*notification*

Invoked whenever the user is about to collapse an item in the outline view. This method is invoked as a result of posting an NSOutlineViewItemWillCollapseNotification

outlineViewItemWillExpand:

– (void)**outlineViewItemWillExpand:**(NSNotification *)*notification*

Invoked whenever the user is about to expand an item in the outline view. This method is invoked as a result of posting an NSOutlineViewItemWillExpandNotification

outlineViewSelectionDidChange:

– (void)**outlineViewSelectionDidChange:**(NSNotification *)*notification*

Invoked immediately after the outline view's selection has changed. This method is invoked as a result of posting an NSOutlineViewSelectionDidChangeNotification.

See also: **NSOutlineViewSelectionDidChangeNotification** (notification)

outlineViewSelectionIsChanging:

– (void)**outlineViewSelectionIsChanging:**(NSNotification *)*notification*

Invoked whenever the outline view's selection changes. This method is invoked as a result of posting an NSOutlineViewSelectionIsChangingNotification.

See also: **NSOutlineViewSelectionIsChangingNotification** (notification)

outlineView:shouldCollapseItem:

– (BOOL)**outlineView:**(NSOutlineView *)*outlineView*
shouldCollapseItem:(id)*item*

Returns YES to permit *outlineView* to collapse *item*, NO to deny permission. The delegate can implement this method to disallow collapsing of specific items.

outlineView:shouldEditTableColumn:item:

– (BOOL)**outlineView:**(NSOutlineView *)*outlineView*
shouldEditTableColumn:(NSTableColumn *)*tableColumn*
item:(id)*item*

Returns YES to permit *outlineView* to edit the cell specified by *tableColumn* and *item*, NO to deny permission. The delegate can implement this method to disallow editing of specific cells.

outlineView:shouldExpandItem:

– (BOOL)**outlineView:**(NSOutlineView *)*outlineView*
shouldExpandItem:(id)*item*

Returns YES to permit *outlineView* to expand *item*, NO to deny permission. The delegate can implement this method to disallow expanding of specific items.

outlineView:shouldSelectItem:

– (BOOL)**outlineView:**(NSOutlineView *)*outlineView*
shouldSelectItem:(id)*item*

Returns YES to permit *outlineView* to select *item*, NO to deny permission. The delegate can implement this method to disallow selection of particular items.

outlineView:shouldSelectTableColumn:

– (BOOL)**outlineView:**(NSOutlineView *)*outlineView*
shouldSelectTableColumn:(NSTableColumn *)*tableColumn*

Returns YES to permit *outlineView* to select *tableColumn*, NO to deny permission. The delegate can implement this method to disallow selection of specific columns.

outlineView:willDisplayCell:forTableColumn:item:

– (void)**outlineView:**(NSOutlineView *)*outlineView*
willDisplayCell:(id)*cell*
forTableColumn:(NSTableColumn *)*tableColumn*
item:(id)*item*

Informs the delegate that *outlineView* is about to display the cell specified by *tableColumn* and *item*. The delegate can modify *cell* to alter its display attributes; for example, making uneditable values display in italic or gray text.

outlineView:willDisplayOutlineCell:forTableColumn:item:

– (BOOL)**outlineView:**(NSOutlineView *)*outlineView*
willDisplayOutlineCell:(id)*cell*
forTableColumn:(NSTableColumn *)*tableColumn*
item:(id)*item*

Informs the delegate that *outlineView* is about to display *cell* (the cell used to draw the expansion symbol) for the column and item specified by *tableColumn* and *item*. The delegate can modify *cell* to alter its display attributes.

selectionShouldChangeInOutlineView:

– (BOOL)**selectionShouldChangeInOutlineView:**(NSOutlineView *)*outlineView*

Returns YES to permit *outlineView* to change its selection (typically a row being edited), NO to deny permission. For example, if the user is editing a cell and enters an improper value, the delegate can prevent the user from selecting or editing any other cells until a proper value has been entered into the original cell. The delegate can implement this method for complex validation of edited rows based on the values of any of their cells.

Notifications

NSOutlineViewColumnDidMoveNotification

Posted whenever a column is moved by user action in the NSOutlineView.

This notification contains a notification object and a userInfo dictionary. The notification object is the NSOutlineView in which a column moved. The userInfo dictionary contains these keys and values:

Key	Value
NSOldColumn	The column's original index (an NSNumber)
NSNewColumn	The column's present index (an NSNumber)

See also: – **moveColumn:toColumn:** (NSTableView)

NSOutlineViewColumnDidResizeNotification

Posted whenever a column is resized in the NSOutlineView.

This notification contains a notification object and a userInfo dictionary. The notification object is the NSOutlineView in which a column was resized. The userInfo dictionary contains these keys and values:

Key	Value
NSOldWidth	The column's original width (an NSNumber)

NSOutlineViewItemDidCollapseNotification

Posted whenever an item is collapsed.

This notification contains a notification object and a userInfo dictionary. The notification object is the NSOutlineView in which an item was collapsed. The userInfo dictionary contains these keys and values:

Key	Value
NSObject	The item that was collapsed (an **id**)

NSOutlineViewItemDidExpandNotification

Posted whenever an item is expanded.

This notification contains a notification object and a userInfo dictionary. The notification object is the NSOutlineView in which an item was expanded. The userInfo dictionary contains these keys and values:

Key	Value
NSObject	The item that was expanded (an **id**)

NSOutlineViewSelectionDidChangeNotification

Posted after the NSOutlineView's selection changes.

This notification contains a notification object but no userInfo dictionary. The notification object is the NSOutlineView whose selection changed.

NSOutlineViewSelectionIsChangingNotification

Posted as the NSOutlineView's selection changes (while the mouse is still down).

This notification contains a notification object but no userInfo dictionary. The notification object is the NSOutlineView whose selection is changing.

NSPageLayout

Inherits From: NSObject

Note: On Mach platforms, NSPageLayout inherits from NSPanel.

Conforms To: NSObject (NSObject)

Declared In: AppKit/NSPageLayout.h

Class Description

NSPageLayout is a panel that queries the user for information such as paper type and orientation. This information is stored in an NSPrintInfo object, and is later used when printing. The NSPageLayout panel is created, displayed, and run (in a modal loop) when a **runPageLayout:** message is sent to the NSApplication object. By default, this message is sent up the responder chain when the user chooses the Page Setup menu item (on Mach platforms the menu item is called Page Layout).

Typically, you access an NSPageLayout panel by invoking the **pageLayout** method. When the class receives a **pageLayout** message, it returns an existing panel rather than create a new one. If a panel is reused, its attributes are reset to the default values so that the effect is the same as receiving a new panel. Because the NSPageLayout object returned by **pageLayout** may be reused, you should only modify it using methods explicitly declared by NSPageLayout. If you must modify an NSPageLayout object in other ways, don't modify the object returned by **pageLayout**; instead, create and manage your own instance using the **alloc...** and **init...** methods.

In most cases it is unnecessary to subclass NSPageLayout—you can customize an NSPageLayout by specifying your own accessory view. You can add your own controls to an NSPageLayout through the **setAccessoryView:** method. The panel is automatically resized to accommodate the NSView that you've added. Note that NSPageLayout does not have accessor methods to obtain the state of its controls. If controls you add through an accessory view need to know the values of existing controls (or vice versa) use the **viewWithTag:** method. You obtain a specific control object by sending **viewWithTag:** to the NSPageLayout object passing one of the following tags (enumerated in AppKit/NSPageLayout.h):

NSPLImageButton
NSPLTitleField
NSPLPaperNameButton
NSPLUnitsButton
NSPLWidthForm
NSPLHeightForm
NSPLOrientationMatrix

NSPLCancelButton
NSPLOKButton

The value can then be obtained by sending an appropriate accessor message to the returned control object.

Method Types

Creating an NSPageLayout

+ pageLayout

Running an NSPageLayout

– runModal
– runModalWithPrintInfo:

Customizing an NSPageLayout

– accessoryView
– setAccessoryView:

Accessing the NSPrintInfo

– printInfo
– readPrintInfo
– writePrintInfo

Updating the display

– convertOldFactor:newFactor:
– pickedButton:
– pickedOrientation:
– pickedPaperSize:
– pickedUnits:

Class Methods

pageLayout

+ (NSPageLayout *)**pageLayout**

Returns a shared NSPageLayout object or a newly created one if it doesn’t already exist.

Instance Methods

accessoryView

– (NSView *)**accessoryView**

Returns the receiver's accessory view (used to customize the receiver).

See also: – **setAccessoryView:**

convertOldFactor:newFactor:

– (void)**convertOldFactor:**(float *)*old*
newFactor:(float *)*new*

This method is for Mach platforms only—it is not defined for other platforms. The standard unit used to measure a paper's dimensions is a point (for example, NSPrintInfo defines a paper's size in points). However, the user can select a different unit of measurement from the NSPageLayout panel. Use this method to get the ratio between a point and the currently selected unit of measurement. Unless this method is invoked by **pickedUnits:** both *old* and *new* will be set to the same ratio value.

The **pickedUnits:** method is invoked when the user selects a new unit of measurement from the NSPageLayout panel. Subclasses should override the **pickedUnits:** method to update any controls, located on the accessory view, that display dimensional values. Use this method to get the old and new ratios. See **pickedUnits:** for details.

pickedButton:

– (void)**pickedButton:**(id)*sender*

This method is for Mach platforms only—it is not defined for other platforms. Invoked when either the OK or Cancel buttons are clicked, and stops the receiver's modal loop. If the OK button was clicked, this method verifies that the height, width and scale entries are acceptable (they must hold positive numbers). If not, the unacceptable entry is selected and the panel isn't stopped. Subclasses should override this method to verify that the controls on the accessory view contain acceptable values.

See also: – **pickedOrientation:**, – **pickedPaperSize:**, – **pickedUnits:**

pickedOrientation:

– (void)**pickedOrientation:**(id)*sender*

This method is for Mach platforms only—it is not defined for other platforms. Invoked when the user selects a page orientation (i.e., portrait or landscape). This method updates the height and width fields, and redraws the paper view.

See also: – **pickedButton:**, – **pickedPaperSize:**, – **pickedUnits:**

pickedPaperSize:

– (void)**pickedPaperSize:**(id)*sender*

This method is for Mach platforms only—it is not defined for other platforms. Invoked when the user selects a paper size from the paper size list. Updates the height and width fields, redraws the paper view, and may switch the portrait/landscape orientation.

See also: – **pickedButton:**, – **pickedOrientation:**, – **pickedUnits:**

pickedUnits:

– (void)**pickedUnits:**(id)*sender*

This method is for Mach platforms only—it is not defined for other platforms. Invoked when the user selects a new unit of measurement from the Units list. The height and width fields are updated.

Subclasses should override this method to update controls in the accessory view that contain unit values. The ratios returned by the **convertOldFactor:newFactor:** method should be used to calculate the new values as shown below, where **myField** is an NSTextField located on the accessory view that needs to be updated:

```
- pickedUnitsLsender
{
    float old, new;

    /* At this point the units have been selected but not set. */
    [self convertOldFactor:&old newFactor:&new];

    /* Update myField based on the conversion factors. */
    [myField setFloatValue:([myField floatValue]*new/old)];

    /* Set the selected units. */
    return [super pickedUnits:sender];
}
```

See also: – **pickedButton:**, – **pickedOrientation:**, – **pickedPaperSize:**

printInfo

– (NSPrintInfo *)**printInfo**

Returns the NSPrintInfo object that is used when the receiver is run (set using the **runModal** or **runModalWithPrintInfo:** methods).

See also: – **readPrintInfo**, – **writePrintInfo**

readPrintInfo

– (void)**readPrintInfo**

Sets the receiver's values to those stored in the NSPrintInfo object used when the receiver is run.

See also: – **printInfo**, – **writePrintInfo**, – **runModal**, – **runModalWithPrintInfo:**

runModal

– (int)**runModal**

Displays the receiver and begins the modal loop. The receiver's values are recorded in the shared NSPrintInfo object. Returns NSCancelButton if the user clicks the Cancel button, otherwise returns NSOKButton.

See also: – **pickedButton:**, – **runModalWithPrintInfo:**

runModalWithPrintInfo:

– (int)**runModalWithPrintInfo:**(NSPrintInfo *)*printInfo*

Displays the receiver and begins the modal loop. The receiver's values are recorded in *printInfo*. Returns NSCancelButton if the user clicks the Cancel button, otherwise returns NSOKButton.

See also: – **pickedButton:**, – **runModal**

setAccessoryView:

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

Adds an NSView to the receiver. Invoke this method to add a custom view containing your controls. The receiver is automatically resized to accommodate *aView*. This method can be invoked repeatedly to change the accessory view depending on the situation. If *aView* is **nil**, then the receiver's current accessory view, if any, is removed.

See also: – **accessoryView**

writePrintInfo

– (void)**writePrintInfo**

Writes the receiver's values to the NSPrintInfo object used when the receiver is run.

See also: – **printInfo**, – **readPrintInfo**, – **runModal**, – **runModalWithPrintInfo:**

NSPanel

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

Class Description

A panel is a special kind of window, typically serving an auxiliary function in an application. NSPanel adds a few special behaviors to NSWindow in support of the role panels play:

- Panels are by default not released when they're closed, since they're usually lightweight and often reused.
- On-screen panels, except for attention panels, are removed from the screen when the application isn't active, and are restored when the application again becomes active. This reduces screen clutter.
- Panels can become the key window, but not the main window.
- If a panel is the key window and has a close button, it closes itself when the user presses the Escape key.

In addition to these automatic behaviors, NSPanel allows you to configure certain other behaviors common to some kinds of panels:

- A panel can be precluded from becoming the key window unless the user clicks in a view that responds to typing. This prevents key window from shifting to the panel unnecessarily. The **setBecomesKeyOnlyIfNeeded:** method controls this behavior.
- Palettes and similar panels can be made to float above standard windows and other panels. This prevents them from being covered and keeps them readily available to the user. The **setFloatingPanel:** method controls this behavior.
- A panel can be made to receive mouse and keyboard events even when another window or panel is being run modally or run in a modal session. This permits actions in the panel to affect the modal window or panel. The **setWorksWhenModal:** method controls this behavior. See "Modal Windows" in the NSWindow class specification for more information on modal windows and panels.

Method Types

Configuring panel behavior

– setFloatingPanel:
– isFloatingPanel
– setBecomesKeyOnlyIfNeeded:
– becomesKeyOnlyIfNeeded
– setWorksWhenModal:
– worksWhenModal

Instance Methods

becomesKeyOnlyIfNeeded

– (BOOL)**becomesKeyOnlyIfNeeded**

Returns YES if the receiver becomes the key window only when the user clicks a view object that needs to be first responder to receive event and action messages; for example if it edits text or otherwise accepts keyboard input. Returns NO if it becomes the key window whenever clicked. NSPanel by default returns NO, indicating that panels become key as other windows do.

See also: – **setBecomesKeyOnlyIfNeeded:**, – **needsPanelToBecomeKey** (NSView)

isFloatingPanel

– (BOOL)**isFloatingPanel**

Returns YES if the receiver is set to float above normal windows, NO otherwise. A floating panel's window level is NSFloatingWindowLevel. NSPanels by default returns NO, indicating that they inhabit the normal window level.

See also: – **setFloatingPanel:**, – **level** (NSWindow)

setBecomesKeyOnlyIfNeeded:

– (void)**setBecomesKeyOnlyIfNeeded:**(BOOL)*flag*

Controls whether the receiver only becomes the key window when the user clicks a view object that edits text or otherwise accepts keyboard input. If *flag* is YES, the receiver only becomes the key window when keyboard input is needed; if *flag* is NO, it becomes the key window whenever clicked. This behavior is by default not set. You should consider setting it only if most controls in the NSPanel aren't text fields, and if

the choices that can be made by entering text can also be made in another way (such as by clicking an item in a pick list).

See also: – **becomesKeyOnlyIfNeeded**, – **needsPanelToBecomeKey** (NSView)

setFloatingPanel:

– (void)**setFloatingPanel:**(BOOL)*flag*

Controls whether the receiver floats above normal windows. If *flag* is YES, sets the receiver's window level to NSFloatingWindowLevel; if *flag* is NO, sets the receiver's window level to NSNormalWindowLevel. The default is NO. It's appropriate for an NSPanel to float above other windows only if all of the following conditions are true:

- It's small enough not to obscure whatever's behind it.
- It's oriented more to the mouse than to the keyboard—that is, if it doesn't become the key window or becomes so only when needed.
- It needs to remain visible while the user works in the application's normal windows; for example, if the user must frequently move the cursor back and forth between a normal window and the panel (such as a tool palette), or if the panel gives information relevant to the user's actions in a normal window.
- It hides when the application is deactivated (the default behavior for panels).

See also: – **isFloatingPanel**, – **setLevel:** (NSWindow)

setWorksWhenModal:

– (void)**setWorksWhenModal:**(BOOL)*flag*

Controls whether the receiver receives keyboard and mouse events even when some other window is being run modally. If *flag* is YES, the application object sends events to the receiver even during a modal loop or session; if *flag* is NO, the receiver gets no events while a modal loop or session is running. See "Modal Windows" in the NSWindow class specification for more information on modal windows and panels.

See also: – **worksWhenModal**, – **runModalForWindow:** (NSApplication), – **runModalSession:** (NSApplication)

worksWhenModal

– (BOOL)**worksWhenModal**

Returns YES if the receiver is able to receive keyboard and mouse events even when some other window is being run modally, NO otherwise. NSPanels by default returns NO, indicating their ineligibility for events

during a modal loop or session. See “Modal Windows” in the NSWindow class specification for more information on modal windows and panels.

See also: – **setWorksWhenModal:**, – **runModalForWindow:** (NSApplication), – **runModalSession:** (NSApplication)

NSParagraphStyle

Inherits From:	NSObject
Conforms To:	NSCoding NSCopying NSMutableCopying NSObject (NSObject)
Declared In:	AppKit/NSParagraphStyle.h

Class Description

NSParagraphStyle and its subclass NSMutableParagraphStyle encapsulate the paragraph or ruler attributes used by the NSAttributedString classes. Instances of these classes are often referred to as *paragraph style objects*, or when no confusion will result, as *paragraph styles*.

A paragraph style object represents a complex attribute value in an attributed string, storing a number of sub-attributes that affect paragraph layout for the characters of the string. Among these sub-attributes are alignment, tab stops, and indents. See the method descriptions for more information on each sub-attribute.

Adopted Protocols

NSCoding	
	– encodeWithCoder: – initWithCoder:
NSCopying	
	– copyWithZone:
NSMutableCopying	
	– mutableCopyWithZone:

Method Types

Creating an NSParagraphStyle	
	+ defaultParagraphStyle

Accessing style information

– alignment
– firstLineHeadIndent
– headIndent
– tailIndent
– tabStops
– lineBreakMode
– maximumLineHeight
– minimumLineHeight
– lineSpacing
– paragraphSpacing

Class Methods

defaultParagraphStyle

+ (NSParagraphStyle *)**defaultParagraphStyle**

Returns the default paragraph style, which contains these values:

Sub-Attribute	Default Value
Alignment	NSNaturalTextAlignment
Tab stops	12 left-aligned tabs, spaced by 28.0 points
Line break mode	NSLineBreakByWordWrapping
All others	0.0

See individual method descriptions for explanations of each sub-attribute.

Instance Methods

alignment

– (NSTextAlignment)**alignment**

Returns the text alignment of the paragraph style, one of:

NSLeftTextAlignment
NSRightTextAlignment
NSCenterTextAlignment

NSJustifiedTextAlignment
NSNaturalTextAlignment

Natural text alignment is realized as left or right alignment depending on the line sweep direction of the first script contained in the paragraph.

See also: – **setAlignment:** (NSMutableParagraphStyle)

firstLineHeadIndent

– (float)**firstLineHeadIndent**

Returns the distance in points from the leading margin of a text container to the beginning of the paragraph's first line. This value is always nonnegative.

See also: – **headIndent**, – **tailIndent**, – **setFirstLineHeadIndent:** (NSMutableParagraphStyle)

headIndent

– (float)**headIndent**

Returns the distance in points from the leading margin of a text container to the beginning of lines other than the first. This value is always nonnegative.

See also: – **firstLineHeadIndent**, – **tailIndent**, – **setHeadIndent:** (NSMutableParagraphStyle)

lineBreakMode

– (NSLineBreakMode)**lineBreakMode**

Returns the mode that should be used to break lines when laying out the paragraph's text. This is one of:

Value	Meaning
NSLineBreakByWordWrapping	Wrapping occurs at word boundaries, unless the word itself doesn't fit on a single line.
NSLineBreakByCharWrapping	Wrapping occurs before the first character that doesn't fit.
NSLineBreakByClipping	Lines are simply not drawn past the edge of the text container.
NSLineBreakByTruncatingHead	Each line is displayed so that the end fits in the container and the missing text is indicated by some kind of ellipsis glyph.
NSLineBreakByTruncatingTail	Each line is displayed so that the beginning fits in the container and the missing text is indicated by some kind of ellipsis glyph.
NSLineBreakByTruncatingMiddle	Each line is displayed so that the beginning and end fit in the container and the missing text is indicated by some kind of ellipsis glyph in the middle.

See also: – **setLineBreakMode:** (NSMutableParagraphStyle)

lineSpacing

– (float)**lineSpacing**

Returns the space in points added between lines within the paragraph (commonly known as *leading*). This value is always nonnegative.

See also: – **maximumLineHeight**, – **minimumLineHeight**, – **paragraphSpacing**, – **setLineSpacing:** (NSMutableParagraphStyle)

maximumLineHeight

– (float)**maximumLineHeight**

Returns the maximum height that any line in the paragraph style will occupy, regardless of the font size or size of any attached graphic. Glyphs and graphics exceeding this height will overlap neighboring lines; however, a maximum height of zero implies no line height limit. This value is always nonnegative. The default value is zero.

Note: Although this limit applies to the line itself, line spacing adds extra space between adjacent lines.

See also: – **minimumLineHeight**, – **lineSpacing**, – **setMaximumLineHeight:** (NSMutableParagraphStyle)

minimumLineHeight

– (float)**minimumLineHeight**

Returns the minimum height that any line in the paragraph style will occupy, regardless of the font size or size of any attached graphic. This value is always nonnegative.

See also: – **maximumLineHeight**, – **lineSpacing**, – **setMinimumLineHeight:** (NSMutableParagraphStyle)

paragraphSpacing

– (float)**paragraphSpacing**

Returns the space added at the end of the paragraph to separate it from the following paragraph. This value is always nonnegative.

See also: – **lineSpacing**, – **setParagraphSpacing:** (NSMutableParagraphStyle)

tabStops

– (NSArray *)**tabStops**

Returns the NSTextTab objects, sorted by location, that define the tab stops for the paragraph style.

See also: – **location** (NSTextTab), – **setTabStops:** (NSMutableParagraphStyle), – **addTabStop:** (NSMutableParagraphStyle), – **removeTabStop:** (NSMutableParagraphStyle)

tailIndent

– (float)**tailIndent**

Returns the distance in points from the margin of a text container to the end of lines. If positive, this is the distance from the leading margin (for example, the left margin in left-to-right text). If zero or negative, it’s the distance from the trailing margin.

For example, a paragraph style designed to fit exactly in a 2-inch wide container has a head indent of 0.0 and a tail indent of 0.0. One designed to fit with a quarter-inch margin has a head indent of 0.25 and a tail indent of –0.25.

See also: – **headIndent**, – **firstLineHeadIndent**, – **setTailIndent:** (NSMutableParagraphStyle)

NSPasteboard

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: AppKit/NSPasteboard.h

Class at a Glance

Purpose

An NSPasteboard object is an interface to a pasteboard server that allows you to transfer data between applications, as in copy, cut, and paste operations. The data can be placed in the pasteboard server in a variety of representations.

Principal Attributes

- Owners
- Data types
- Change count
- Name

Creation

+ generalPasteboard
+ pasteboardWithName:

Commonly Used Methods

– types	Returns an NSArray of pasteboard data types.
– declareTypes:owner:	Prepares NSPasteboard to receive new data.
– dataForType:	Reads data from a pasteboard.
– setData:forType:	Writes data to a pasteboard.
– stringForType:	Reads an NSString from a pasteboard.
– setStringForType:	Writes an NSString to a pasteboard.

Class Description

NSPasteboard objects transfer data to and from the pasteboard server. The server is shared by all running applications. It contains data that the user has cut or copied, as well as other data that one application wants to transfer to another. NSPasteboard objects are an application's sole interface to the server and to all pasteboard operations.

Named Pasteboards

Data in the pasteboard server is associated with a name that indicates how it's to be used. Each set of data and its associated name is, in effect, a separate pasteboard, distinct from the others. An application keeps a separate NSPasteboard object for each named pasteboard that it uses. There are five standard pasteboards in common use:

Pasteboard	Description
General pasteboard	The pasteboard that's used for ordinary cut, copy, and paste operations. It holds the contents of the last selection that's been cut or copied.
Font pasteboard	The pasteboard that holds font and character information and supports Copy Font and Paste Font commands that my be implemented in a text editor.
Ruler pasteboard	The pasteboard that holds information about paragraph formats in support of the Copy Ruler and Paste Ruler commands that may be implemented in a text editor.
Find pasteboard	The pasteboard that holds information about the current state of the active application's Find panel. This information permits users to enter a search string into the Find panel, then switch to another application to conduct another search.
Drag pasteboard	The pasteboard that stores data to be moved as the result of a drag operation.

Each standard pasteboard is identified by a unique name (stored in global string objects):

- NSGeneralPboard
- NSFontPboard
- NSRulerPboard
- NSFindPboard
- NSDragPboard

You can create private pasteboards by asking for an NSPasteboard object with any name other than those listed above. Data in a private pasteboard may then be shared by passing its name between applications.

The NSPasteboard class makes sure there's never more than one object for each named pasteboard on the computer. If you ask for a new object when one has already been created for the pasteboard with that name, the existing object will be returned.

Data Types

Data can be placed in the pasteboard server in more than one representation. For example, an image might be provided both in Tag Image File Format (TIFF) and as encapsulated PostScript code (EPS). Multiple representations give pasting applications the option of choosing which data type to use. In general, an application taking data from the pasteboard should choose the richest representation it can handle—rich text over plain ASCII, for example. An application putting data in the pasteboard should promise to supply it in as many data types as possible, so that as many different applications as possible can use it.

Filtering services transform the data from one representation to another. Typically, these services aren't invoked until data is read from a pasteboard.

Data types are identified by NSString objects containing the full type name. These global variables identify the string objects for the standard pasteboard types:

Type	Description
NSColorPboardType	NSColor data
NSFileContentsPboardType	A representation of a file's contents
NSFilenamesPboardType	NSString designating one or more file names
NSFontPboardType	Font and character information
NSPostScriptPboardType	Encapsulated PostScript code (EPS)
NSRulerPboardType	Paragraph formatting information
NSRTFPboardType	Rich Text Format (RTF)
NSRTFDPboardType	RTFD formatted file contents
NSSelectionPboardType	Describes a selection for use with data linking
NSStringPboardType	NSString data
NSTabularTextPboardType	NSString containing tab-separated fields of text
NSTIFFPboardType	Tag Image File Format (TIFF)

Typically, data is written to the pasteboard using **setData:forType:** and read using **dataForType:**. Some of these types can only be written with certain methods. For instance, NSFilenamesPboardType's form is an array of NSStrings and requires special handling. Use these methods to write these types:

Type	Writing Method	Reading Method
NSColorPboardType	NSColor class methods	NSColor class methods
NSFileContentsPboardType	**writeFileContents:**	**readFileContentsType:toFile:**
NSFilenamesPboardType	**setPropertyList:forType:**	**propertyListForType:**
NSStringPboardType	**setString:forType:**	**stringForType:**

You don't have to write the data (using **setData:forType:**) in all types that you've declared for the pasteboard: This avoids unneeded conversions. If data is requested from a pasteboard in a format that's not present, the owner of the pasteboard receives a **pasteboard:provideDataForType:** message notifying it that it needs to supply the data in that format. It then supplies data in the requested type by invoking one of the **setData:forType:**, **setString:forType:**, or **setPropertyList:forType:** methods on the pasteboard.

The class methods **pasteboardByFilteringData:ofType:**, **pasteboardByFilteringFile:**, and **pasteboardByFilteringTypesInPasteboard:** return a pasteboard with data that is filtered into all types derivable from the current types using available filter services. (For more information on filter services see **/NextLibrary/Documentation/NextDev/TasksAndConcepts/ProgrammingTopics/Services.rtf**.) The pasteboards returned by these methods are autoreleased instances of NSPasteboard.

Types other than those listed above can also be used. For example, your application may keep data in a private format that's richer than any of the existing types. That format can also be used as a pasteboard type.

Reading and Writing RTFD Data

The NSRTFDPboardType is used for the contents of an RTFD file package (a directory containing an RTF text file and one or many EPS and TIFF image files). There are several ways to work with RTFD data. If you have an NSFileWrapper object that represents an RTFD file wrapper, you can send it the **serializedRepresentation** method to return the RTFD data and write that to the pasteboard as follows:

```
NSFileWrapper *tempRTFDData = [[[NSFileWrapper alloc]
    initWithPath:@"/tmp/foo.rtfd"] autorelease];
[pboard setData:[tempRTFDData serializedRepresentation]
    forType:NSRTFDPboardType];
```

In addition to NSFileWrapper, classes such as NSAttributedString and NSText can return RTFD data. If you're using one of these classes, you would do the following to write RTFD data to the pasteboard:

```
NSAttributedString *attrString;
...
[pboard setData:[attrString RTFDFromRange:NSMakeRange(0, [attrString length])]
    forType:NSRTFDPboardType];
```

Change Count

The change count is a computer-wide variable that increments every time the contents of the pasteboard changes (a new owner is declared). An independent change count is maintained for each named pasteboard. By examining the change count, an application can determine whether the current data in the pasteboard is the same as the data it last received.

The **changeCount**, **addTypes:owner:**, and **declareTypes:owner:** methods return the change count. A **types** or **availableTypeFromArray:** message should be sent by the pasteboard before reading data so that the change count is valid.

Errors

Except where errors are specifically mentioned in the method descriptions, any communications error with the pasteboard server raises an NSPasteboardCommunicationException.

Method Types

Creating and releasing an NSPasteboard object

+ generalPasteboard
+ pasteboardByFilteringData:ofType:
+ pasteboardByFilteringFile:
+ pasteboardByFilteringTypesInPasteboard:
+ pasteboardWithName:
+ pasteboardWithUniqueName
+ typesFilterableTo:
– releaseGlobally

Referring to a pasteboard by name

– name

Writing data

– addTypes:owner:
– declareTypes:owner:
– setData:forType:
– setPropertyList:forType:
– setString:forType:
– writeFileContents:
– writeFileWrapper:

Determining Types

– availableTypeFromArray:
– types

Reading Data

– changeCount
– dataForType:
– propertyListForType:
– readFileContentsType:toFile:
– readFileWrapper
– stringForType:

Methods Implemented by the Owner

– pasteboardChangedOwner:
– pasteboard:provideDataForType:

Class Methods

generalPasteboard

+ (NSPasteboard *)**generalPasteboard**

Returns the general NSPasteboard. This invokes **pasteboardWithName:** to obtain the pasteboard.

pasteboardByFilteringData:ofType:

+ (NSPasteboard *)**pasteboardByFilteringData:**(NSData *)*data* **ofType:**(NSString *)*type*

Creates and returns a new pasteboard with a unique name that supplies *data* in as many types as possible given the available filter services. The returned pasteboard also declares data of the supplied type *type*.

No filter service is invoked until the data is actually requested, so invoking this method is reasonably inexpensive.

pasteboardByFilteringFile:

+ (NSPasteboard *)**pasteboardByFilteringFile:**(NSString *)*filename*

Creates and returns a new pasteboard with a unique name that supplies the data in *filename* in as many types as possible given the available filter services. No filter service is invoked until the data is actually requested, so invoking this method is reasonably inexpensive.

pasteboardByFilteringTypesInPasteboard:

+ (NSPasteboard *)**pasteboardByFilteringTypesInPasteboard:**(NSPasteboard *)*pasteboard*

Creates and returns a new pasteboard with a unique name that supplies the data on *pasteboard* in as many types as possible given the available filter services. This process can be thought of as expanding the pasteboard, since the new pasteboard generally will contain more representations of the data than *pasteboard*.

This method returns *pasteboard* if *pasteboard* was returned by one of the **pasteboardByFiltering...** methods, so a pasteboard can't be expanded multiple times. This method only returns the original types and the types that can be created as a result of a single filter; the pasteboard will not have defined types that are the result of translation by multiple filters.

No filter service is invoked until the data is actually requested, so invoking this method is reasonably inexpensive.

pasteboardWithName:

+ (NSPasteboard *)**pasteboardWithName:**(NSString *)*name*

Returns the pasteboard for the *name* pasteboard. A new object is created only if the application doesn't yet have a NSPasteboard object for the specified name; otherwise, the existing one is returned. To get a standard pasteboard, *name* should be one of the following variables:

NSGeneralPboard
NSFontPboard
NSRulerPboard
NSFindPboard
NSDragPboard

Other names can be assigned to create private pasteboards for other purposes.

pasteboardWithUniqueName

+ (NSPasteboard *)**pasteboardWithUniqueName**

Creates and returns a new pasteboard with a name that is guaranteed to be unique with respect to other pasteboards on the computer. This method is useful for applications that implement their own interprocess communication using pasteboards.

typesFilterableTo:

+ (NSArray *)**typesFilterableTo:**(NSString *)*type*

Returns an autoreleased array listing the types of data that *type* can be converted to by available filter services. The array contains the original type.

Instance Methods

addTypes:owner:

– (int)**addTypes:**(NSArray *)*newTypes* **owner:**(id)*newOwner*

Adds the data types in *newTypes* to the NSPasteboard and declares a new owner *newOwner*. This method can be useful when multiple entities (such as a combination of application and library methods) contribute data for a single copy command. It should only be invoked after a **declareTypes:owner:** message has been sent for the same types. The owner for the new types may be different from the owner(s) of the previously declared types.

Returns the new change count, or 0 in case of an error.

See also: **– changeCount**

availableTypeFromArray:

– (NSString *)**availableTypeFromArray:**(NSArray *)*types*

Scans the types defined by *types* and returns the first type that matches a type declared on the receiving NSPasteboard.

A **types** or **availableTypeFromArray:** message should be sent before reading any data from the NSPasteboard.

changeCount

– (int)**changeCount**

Returns the NSPasteboard's change count.

See also: **– declareTypes:owner:**

dataForType:

– (NSData *)**dataForType:**(NSString *)*dataType*

Reads the *dataType* representation of the current contents of the NSPasteboard. *dataType* should be one of the types returned by the **types** method. A **types** or **availableTypeFromArray:** message should be sent before invoking **dataForType:**.

If the data is successfully read, this method returns the data. It returns **nil** if the contents of the pasteboard have changed (if the change count has been incremented by a **declareTypes:owner:** message) since they were last checked with the **types** method. It also returns **nil** if the pasteboard server can't supply the data in time—for example, if the NSPasteboard's owner is slow in responding to a **pasteboard: provideDataForType:** message and the interprocess communication times out. All other errors raise an NSPasteboardCommunicationException exception.

If **nil** is returned, the application should put up a panel informing the user that it was unable to carry out the paste operation.

The NSData object that this method returns is autoreleased.

declareTypes:owner:

– (int)**declareTypes:**(NSArray *)*newTypes* **owner:**(id)*newOwner*

Prepares the NSPasteboard for a change in its contents by declaring the new types of data it will contain and a new owner. This is the first step in responding to a user's copy or cut command and must precede the messages that actually write the data. A **declareTypes:owner:** message essentially changes the contents of the pasteboard: It invalidates the current contents of the pasteboard and increments its change count.

newTypes is an array of NSStrings that name types the new contents of the pasteboard may assume. The types should be ordered according to the preference of the source application, with the most preferred type coming first (typically, the richest representation).

The *newOwner* is the object responsible for writing data to the pasteboard in all the types listed in *newTypes*. You can write the data immediately after declaring the types, or wait until it's required for a paste operation. If you wait, the owner will receive a **pasteboard:provideDataForType:** message requesting the data in a particular type when it's needed. You might choose to write data immediately for the most preferred type, but wait for the others to see whether they'll be requested.

The *newOwner* can be NULL if data is provided for all types immediately. Otherwise, the owner should be an object that won't be released. It should not, for example, be the NSView that displays the data if that NSView is in a window that might be closed.

Returns the pasteboard's new change count.

See also: – **setString:forType:**, – **addTypes:owner:**, – **changeCount**

name

– (NSString *)**name**

Returns the NSPasteboard's name.

See also: **+ pasteboardWithName:**

propertyListForType:

– (id)**propertyListForType:**(NSString *)*dataType*

Returns a *property list* object using the type specified by *dataType*.

A property list is an object of the NSArray, NSData, NSDictionary, or NSString classes—or any combination thereof.

A **types** or **availableTypeFromArray:** message should be sent before invoking **propertyListForType:**.

This method invokes **dataForType:**.

See also: **– setPropertyList:forType:**

readFileContentsType:toFile:

– (NSString *)**readFileContentsType:**(NSString *)*type* **toFile:**(NSString *)*filename*

Reads data representing a file's contents from the NSPasteboard, and writes it to the file *filename*. An **availableTypeFromArray:** or **types** message should be sent before invoking **readFileContentsType: toFile:**.

Data of any file contents *type* should only be read using this method. *type* should generally be specified; if *type* is NULL, a type based on filename's extension (as returned by the **NSCreateFileContentsPboardType** function) is substituted. If data matching *type* isn't found on the NSPasteboard, data of type NSFileContentsPboardType is requested. Returns the name of the file that the data was actually written to.

See also: **– writeFileContents:**

readFileWrapper

– (NSFileWrapper *)**readFileWrapper**;

Reads data representing a file's contents from the NSPasteboard and returns it as a file wrapper. If there is no data of type NSFileContentsPboardType on the pasteboard, this method returns NULL.

releaseGlobally

– (void)**releaseGlobally**

Releases the NSPasteboard's resources in the pasteboard server. Since an NSPasteboard object is an autoreleased instance of NSPasteboard, it isn't released by this method, and its retain count isn't changed.

After this method is invoked, no other application will be able to use the named NSPasteboard. A temporary, privately named pasteboard can be released this way when it's no longer needed, but a standard NSPasteboard should never be released globally.

setData:forType:

– (BOOL)**setData:**(NSData *)*data* **forType:**(NSString *)*dataType*

Writes data to the pasteboard server. *dataType* gives the type of data being written; it must be a type that was declared in the previous **declareTypes:owner:** message. *data* points to the data to be sent to the pasteboard server.

Returns YES if the data is successfully written or returns NO if ownership of the pasteboard has changed. Any other error raises an NSPasteboardCommunicationException.

See also: **– setPropertyList:forType:**, **– setString:forType:**

setPropertyList:forType:

– (BOOL)**setPropertyList:**(id)*propertyList* **forType:**(NSString *)*dataType*

Writes data to the pasteboard server. *dataType* gives the type of data being written; it must be a type that was declared in the previous **declareTypes:owner:** message. *propertyList* points to the data to be sent to the pasteboard server.

This method invokes **setData:forType:** with a serialized property list parameter.

Returns YES if the data is successfully written or returns NO if ownership of the pasteboard has changed. Any other error raises an NSPasteboardCommunicationException.

See also: **– setString:forType:**

setString:forType:

– (BOOL)**setString:**(NSString *)*string* **forType:**(NSString *)*dataType*

Writes data to the pasteboard server. *dataType* gives the type of data being written; it must be a type that was declared in the previous **declareTypes:owner:** message. *string* points to the data to be sent to the pasteboard server.

This method invokes **setPropertyList:forType:** to perform the write.

Returns YES if the data is successfully written or returns NO if ownership of the pasteboard has changed. Any other error raises an NSPasteboardCommunicationException.

See also: – **setData:forType:**, – **setString:forType:**

stringForType:

– (NSString *)**stringForType:**(NSString *)*dataType*

Returns an NSString using the type specified by *dataType*. A **types** or **availableTypeFromArray:** message should be sent before invoking **stringForType:**.

This method invokes **propertyListForType:** to obtain the string.

types

– (NSArray *)**types**

Returns an array of the NSPasteboard's data types.

Returns an array of the types that were declared for the current contents of the NSPasteboard. The array is an array of NSStrings holding the type names. Types are listed in the same order that they were declared.

A **types** or **availableTypeFromArray:** message should be sent before reading any data from the NSPasteboard.

See also: **– declareTypes:owner:**, **– dataForType:**

writeFileContents:

– (BOOL)**writeFileContents:**(NSString *)*filename*

Writes the contents of the file *filename* to the NSPasteboard object and declares the data to be of type NSFileContentsPboardType and also of a type appropriate for the file's extension (as returned by the **NSCreateFileContentsPboardType** function when passed the files extension), if it has one. Returns YES if the data from *filename* was successfully written to the NSPasteboard and NO otherwise.

See also: **– readFileContentsType:toFile:**

writeFileWrapper:

– (BOOL)**writeFileWrapper:**(NSFileWrapper *)*wrapper*;

Writes the serialized contents of the filewrapper *wrapper* to the NSPasteboard object, and declares the data to be of type NSFileContentsPboardType and also of a type appropriate for the file's extension (as returned by the **NSCreateFileContentsPboardType** function when passed the file's extension), if it has one. If *wrapper* does not have a preferred filename, this method raises an exception. Returns YES if it could successfully write the data from *wrapper* to the NSPasteboard; and returns NO, otherwise.

Methods Implemented by the Owner

pasteboardChangedOwner:

– (void)**pasteboardChangedOwner:**(NSPasteboard *)*sender*

Notifies a prior owner of the *sender* pasteboard (and owners of representations on the pasteboard) that the pasteboard has changed owners. This method is optional and need only be implemented by pasteboard owners that need to know when they have lost ownership. The owner is not able to read the contents of the pasteboard when responding to this method. The owner should be prepared to receive this method at any time, even from within the **declareTypes:owner:** used to declare ownership.

See also: **– changeCount**

pasteboard:provideDataForType:

– (void)**pasteboard:**(NSPasteboard *)*sender*
provideDataForType:(NSString *)*type*

Implemented by the owner (previously declared in a **declareTypes:owner:** message) to provide promised data. The owner receives a **pasteboard:provideDataForType:** message from the sender pasteboard when the data is required for a paste operation; *type* gives the type of data being requested. The requested data should be written to *sender* using the **setData:forType:**, **setPropertyList:forType:**, or **setString:forType:** methods.

pasteboard:provideDataForType: messages may also be sent to the owner when the application is shut down through Application's **terminate:** method. This is the method that's invoked in response to a Quit command. Thus the user can copy something to the pasteboard, quit the application, and still paste the data that was copied.

A **pasteboard:provideDataForType:** message is sent only if *type* data hasn't already been supplied. Instead of writing all data types when the cut or copy operation is done, an application can choose to implement this method to provide the data for certain types only when they're requested.

If an application writes data to the NSPasteboard in the richest, and therefore most preferred, type at the time of a cut or copy operation, its **pasteboard:provideDataForType:** method can simply read that data from the pasteboard, convert it to the requested *type*, and write it back to the pasteboard as the new type.

NSPICTImageRep

Inherits From: NSImageRep : NSObject

Conforms To: NSCoding (from NSImageRep)
NSCopying (from NSImageRep)
NSObject (from NSObject)

Declared In: AppKit/NSPICTImageRep.h

Class Description

An NSPICTImageRep is an object that can render an image from a PICT format data stream as described in *Inside Macintosh: Imaging With QuickDraw*. This includes PICT format version 1, version 2, and extended version 2 pictures.

Warning: There is no guarantee that the image will render exactly the same as it would under QuickDraw because of the differences between Display PostScript and QuickDraw. In particular, some transfer modes and region operations may not be supported.

Like most other kinds of NSImageReps, an NSPICTImageRep is generally used indirectly, through an NSImage object. An NSImage must be able to choose from among various representations of a given image. It also needs to provide an off-screen cache of the appropriate depth for any image it uses. It determines this information by querying its NSImageReps.

Thus to work with an NSImage, an NSPICTImageRep must be able to provide some information about its image. The bounding box is obtained from the PICT format data. Use these methods, inherited from the NSImageRep class, to set the other attributes of an NSPICTImageRep object:

setColorSpaceName:
setAlpha:
setPixelsHigh:
setPixelsWide:
setBitsPerSample:

Note that if these attributes aren't set, and an NSPICTImageRep is used in an NSImage with other representations, NSImage won't be able to select from among the representations. In actual practice, this usually isn't a problem.

Method Types

Creating an NSPICTImageRep

+ imageRepWithData:
– initWithData:

Getting image data

– boundingBox
– PICTRepresentation

Class Methods

imageRepWithData:

+ (id)**imageRepWithData:**(NSData *)*pictData*

Creates a new NSPICTImageRep instance and then invokes **initWithData:** to initialize it with the contents of *pictData*, a PICT format data stream. If the new object can't be initialized for any reason (for example, *pictData* doesn't conform to the PICT file format), this method frees the receiver and returns **nil**. Otherwise, it returns a new instance of NSPICTImageRep.

See also: – **PICTRepresentation**

Instance Methods

boundingBox

– (NSRect)**boundingBox**

Returns the rectangle that bounds the image. The rectangle is obtained from the PICT format data, specifically the *picFrame* field in the picture header. See *Inside Macintosh: Imaging With QuickDraw* for information on the picture header.

initWithData:

– (id)**initWithData:**(NSData *)*pictData*

Initializes the receiver, a newly allocated NSPICTImageRep object, with the contents of *pictData*, a PICT format data stream. If *pictData* is obtained directly from a PICT file or document, it contains a 512-byte header before the actual picture data starts. This method simply ignores that header. If the new object can't be initialized for any reason (for example, *pictData* doesn't conform to the PICT file format), this method frees the receiver and returns **nil**. Otherwise, it returns **self**.

See also: + **imageRepWithData:**, – **PICTRepresentation**

PICTRepresentation

– (NSData *)**PICTRepresentation**

Returns the PICT representation of the receiver. The returned PICT data is a copy of the data supplied to **initWithData:** minus the 512 byte header if it is present. Note, PICT files or documents contain a 512-byte header, so if you wish to save the returned data to a file you need to precede the data with 512 bytes (all zero) to conform to the PICT document format.

NSPopUpButton

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

Class at a Glance

Purpose

An NSPopUpButton object controls a pop-up list or a pull-down list, from which a user can select an item.

Principal Attribute

A list of objects that conform to the NSMenuItem protocol.

Creation

Interface Builder

Commonly Used Methods

– selectedItem	Returns the currently selected item.
– indexOfSelectedItem	Returns an integer identifying the currently selected item.
– titleOfSelectedItem	Returns a string identifying the currently selected item.

Class Description

The NSPopUpButton class defines objects that implement the pop-up and pull-down lists of the OpenStep graphical user interface. You normally create an NSPopUpButton using Interface Builder.

When configured to display a pop-up list, an NSPopUpButton contains a number of options and usually displays the option that was last selected. A pop-up list is often used for selecting items from a small- to

medium-sized set of options (like the zoom factor for a document window). It's a useful alternative to a matrix of radio buttons or an NSBrowser or NSTableView when screen space is at a premium; a zoom factor pop-up can easily fit next to a scroll bar at the bottom of a window, for example.

When configured to display a pull-down list, an NSPopUpButton is generally used for selecting commands in a very specific context. You can think of a pull-down list as a compact form of menu. A pull-down list's display isn't affected by the user's actions, and a pull-down list usually displays the first item in the list. When the commands only make sense in the context of a particular display, a pull-down list can be used in that display to keep the related actions nearby and to keep them out of the way when that display isn't visible.

The items in the pop-up list or pull-down list (referred to simply as a list in this class description) are objects that conform to the NSMenuItem protocol. Thus, you can send any message defined in that protocol to any item in the list.

Using an NSPopUpButton

Although the NSPopUpButton class defines an initialization method and methods that set up the list created by this class, you usually don't invoke these methods in your program. The typical way to create an NSPopUpButton is to use Interface Builder. You can define the NSPopUpButton's target and action, as well as the targets and actions of each item in the NSPopUpButton's list, programmatically or through Interface Builder. The NSPopUpButton methods you use most often are the methods that tell you which item is selected.

For example, suppose you want to create a pop-up list from which your user may select a language. You want your controller object to set an instance variable named **language** to an enum constant that corresponds to the value that the user has chosen. You use Interface Builder to create the NSPopUpButton object, name it (**languagePopUp** in this example) add items to it, and configure it as a pop-up list. The actual code you write might look like this:

```
typedef enum _languageValue {
    English,
    French,
    German
} languageValue;

- (void)setLanguage:(id)sender{
    NSString *title = [languagePopUp titleOfSelectedItem];
    if ([title isEqualToString:@"English"])
        language = English;
    else if ([title isEqualToString:@"French"])
        language = French;
    else if ([title isEqualToString:@"German"])
        language = German;
}
```

Method Types

Initializing an NSPopUpButton

– initWithFrame:pullsDown:

Setting the type of list

– setPullsDown:
– pullsDown
– setAutoenablesItems:
– autoenablesItems

Inserting and deleting Items

– addItemWithTitle:
– addItemsWithTitles:
– insertItemWithTitle:atIndex:
– removeAllItems
– removeItemWithTitle:
– removeItemAtIndex:

Getting the user’s selection

– selectedItem
– titleOfSelectedItem
– indexOfSelectedItem

Setting the current selection

– selectItem:
– selectItemAtIndex:
– selectItemWithTitle:

Getting menu items

– numberOfItems
– itemArray
– itemAtIndex:
– itemTitleAtIndex:
– itemTitles
– itemWithTitle:
– lastItem

Getting the indices of menu items

– indexOfItem:
– indexOfItemWithTag:
– indexOfItemWithTitle:
– indexOfItemWithRepresentedObject:
– indexOfItemWithTarget:andAction:

Setting the cell edge to pop out from in restricted situations

– preferredEdge
– setPreferredEdge:

Setting the font

– setFont:

Setting the title

– setTitle:

Setting the state

– synchronizeTitleAndSelectedItem

Instance Methods

addItemWithTitle:

– (void)**addItemWithTitle:**(NSString *)*title*

Adds an item named *title* to the end of the list. This method then calls **synchronizeTitleAndSelectedItem** to make sure that the item displayed matches the currently selected item.

See also: – **insertItemWithTitle:atIndex:**, – **removeItemWithTitle:**, – **setTitle:**

addItemsWithTitles:

– (void)**addItemsWithTitles:**(NSArray *)*itemTitles*

Adds multiple items to the end of the list. The titles for the new items are taken from the *itemTitles* array. Once the items are added, this method uses **synchronizeTitleAndSelectedItem** to make sure that the item displayed matches the currently selected item.

See also: – **insertItemWithTitle:atIndex:**, – **removeAllItems**, – **removeItemWithTitle:**

autoenablesItems

– (BOOL)**autoenablesItems**

Returns whether the NSPopUpButton automatically enables and disables its items every time a user event occurs. Autoenabling is turned on unless you send the message **setAutoenablesItems:NO** to the NSPopUpButton. See the NSMenuActionResponder informal protocol for more information on autoenabling menu items.

See also: – **setAutoenablesItems:**

indexOfItem:

– (int)**indexOfItem:**(id <NSMenuItem>)*anObject*

Returns the index of menu item *anObject* in the pop-up list or -1 if the menu item is not found. This method invokes the method of the same name of its NSPopUpButtonCell.

indexOfItemWithRepresentedObject:

– (int)**indexOfItemWithRepresentedObject:**(id)*anObject*

Returns the index of the first menu item in the pop-up list that holds the represented object *anObject*, or -1 if no menu item with this object is found. Represented objects bear some direct relation to the title or image of a menu item; for example, an item entitled “100” might have an NSNumber encapsulating that value as its represented object. This method invokes the method of its NSPopUpButtonCell that has the same name.

indexOfItemWithTag:

– (int)**indexOfItemWithTag:**(int)*tag*

Returns the index of the first menu item in the pop-up list that has the tag value *tag* or -1 if the item is not found. This method invokes the method of the same name of its NSPopUpButtonCell.

indexOfItemWithTarget:andAction:

– (int)**indexOfItemWithTarget:**(id)*target* **andAction:**(SEL)*actionSelector*

Returns the index of the first menu item in the pop-up list that has the target of *target* and the action *actionSelector*. If *actionSelector* is NULL, the index of the first menu item in the pop-up list that has a target of *target* is returned. If no menu item matching the above criteria is found, -1 is returned. This method invokes the method of the same name of its NSPopUpButtonCell.

indexOfItemWithTitle:

– (int)**indexOfItemWithTitle:**(NSString *)*title*

Returns the index of the first item whose title matches *title* or –1 if no match is found.

indexOfSelectedItem

– (int)**indexOfSelectedItem**

Returns the index of the item last selected by the user or –1 if there's no selected item.

See also: – **selectedItem**, – **titleOfSelectedItem**

initWithFrame:pullsDown:

– (id)**initWithFrame:**(NSRect)*frameRect* **pullsDown:**(BOOL)*flag*

Initializes a newly allocated NSPopUpButton, giving it the dimensions specified by *frameRect*. If *flag* is YES, the receiver is initialized to operate as a pull-down list; otherwise, it operates as a pop-up list. I

See also: – **pullsDown**, – **setPullsDown:**

insertItemWithTitle:atIndex:

– (void)**insertItemWithTitle:**(NSString *)*title* **atIndex:**(int)*index*

Inserts an item with the name *title* at position *index* in the list. Index 0 indicates the top item. Once the item is inserted, this method uses **synchronizeTitleAndSelectedItem** to make sure that the item displayed matches the currently selected item.

If an item with the name *title* already exists in the list, it's removed and the new one is added. This essentially moves *title* to a new position. If you want to move an item, it's better to invoke **removeItemWithTitle:** explicitly and then send this method.

See also: **– addItemWithTitle:**, **– addItemsWithTitles:**, **– indexOfItemWithTitle:**, **– removeItemWithTitle:**

itemArray

– (NSArray *)**itemArray**

Returns the NSArray that holds the list's items. The NSPopUpButton's list is actually an NSArray of objects conforming to the NSMenuItem protocol. Usually you access the list's items and modify the list by sending messages directly to the NSPopUpButton rather than accessing the NSArray.

See also: – **itemAtIndex:**, – **insertItemWithTitle:atIndex:**, – **removeItemAtIndex:**

itemAtIndex:

– (id <NSMenuItem>)**itemAtIndex:**(int)*index*

Returns the list item at *index*. If there is no item at *index*, this method returns **nil**.

See also: – **itemWithTitle:**, – **lastItem**

itemTitleAtIndex:

– (NSString *)**itemTitleAtIndex:**(int)*index*

Returns the title of the item at *index*. If there is no item at *index*, this method returns the empty string.

See also: – **itemTitles**

itemTitles

– (NSArray *)**itemTitles**

Returns an NSArray object that holds the titles of all of the items in the list. The titles appear in the order in which the items appear in the list.

See also: – **itemTitleAtIndex:**, – **itemWithTitle:**, – **numberOfItems**

itemWithTitle:

– (id <NSMenuItem>)**itemWithTitle:**(NSString *)*title*

Returns the first item whose title is *title*. If there is no item with this title, this method returns **nil**.

See also: – **addItemWithTitle:**, – **selectItemWithTitle:**, – **itemAtIndex:**, – **indexOfItemWithTitle:**

lastItem

– (id <NSMenuItem>)**lastItem**

Returns the last item in the list.

See also: – **itemAtIndex:**

numberOfItems

– (int)**numberOfItems**

Returns the number of items in the list.

See also: – **lastItem**

preferredEdge

– (NSRectEdge)**preferredEdge**

Returns the edge of the receiver next to which the pop-up list is displayed under restrictive screen conditions. For pull-down lists, the default behavior is to display the list under the receiver. For most pop-up lists, NSPopUpButton attempts to show the selected item directly over the button.

See also: – **setPreferredEdge:**

pullsDown

– (BOOL)**pullsDown**

Returns YES if the receiver is configured as a pull-down list or NO if it's configured as a pop-up list.

See also: – **setPullsDown:**

removeAllItems

– (void)**removeAllItems**

Removes all items in the receiver's item list. This method then uses **synchronizeTitleAndSelectedItem** to refresh the list.

See also: – **numberOfItems**, – **removeItemAtIndex:**, – **removeItemWithTitle:**

removeItemAtIndex:

– (void)**removeItemAtIndex:**(int)*index*

Removes the item at *index*. This method then uses **synchronizeTitlesAndSelectedItem** to make sure the title displayed matches the currently selected item.

See also: – **insertItemWithTitle:atIndex:**, – **removeAllItems**, – **removeItemWithTitle:**

removeItemWithTitle:

– (void)**removeItemWithTitle:**(NSString *)*title*

Removes the first item named *title*. This method then uses **synchronizeTitleAndSelectedItem** to refresh the list.

See also: – **addItemWithTitle:**, – **removeAllItems**, – **removeItemAtIndex:**

selectedItem

– (id <NSMenuItem>)**selectedItem**

Returns the item last selected by the user (the item that was highlighted when the user released the mouse button). If there is no selected item, this method returns **nil**. It is possible for a pull-down list's selected item to be its first item.

selectItem:

– (void)**selectItem**(id <NSMenuItem>**)***anObject*

Selects the menu item *anObject* in the pop-up list. If *anObject* is **nil**, all items in the list are deselected (this is a technique for obtaining a pop-up list with no items selected).

selectItemAtIndex:

– (void)**selectItemAtIndex:**(int)*index*

Selects the item in the list at *index* and invokes **synchronizeTitleAndSelectedItem** to make sure the title displayed matches the currently selected item. If index is -1 all items in the list are deselected.

See also: – **indexOfSelectedItem**

selectItemWithTitle:

– (void)**selectItemWithTitle:**(NSString *)*title*

Selects the first item with the given *title* and invokes **synchronizeTitleAndSelectedItem** to make sure the title displayed matches the currently selected item.

See also: – **indexOfItemWithTitle:**, – **addItemWithTitle:**, – **setTitle:**

setAutoenablesItems:

– (void)**setAutoenablesItems:**(BOOL)*flag*

Sets whether the NSPopUpButton automatically enables and disables its items every time a user event occurs. Autoenabling is turned on unless you specify NO as the value for *flag*. See the NSMenuActionResponder informal protocol for more information on autoenabling menu items.

See also: – **autoenablesItems**

setFont:

– (void)**setFont:**(NSFont *)*fontObject*

Sets the font used for the items' titles to *fontObject*. The NSPopUpButton invalidates its display at this point, but since it normally won't be on the screen when it receives this message, this shouldn't cause any undesirable side-effects.

setPreferredEdge:

– (void)**setPreferredEdge:**(NSRectEdge)*edge*

Sets the edge of the receiver next to which the pop-up list should appear under restrictive screen conditions. For pull-down lists, the default behavior is to display the list under the receiver. For most pop-up lists, NSPopUpButton attempts to show the selected item directly over the button.

See also: – **preferredEdge**

setPullsDown:

– (void)**setPullsDown:**(BOOL)*flag*

If *flag* is YES, the receiver is configured as a pull-down list. If *flag* is NO, the receiver is configured as a pop-up list.

See also: – **initWithFrame:pullsDown:**, – **pullsDown**

setTitle:

– (int)**setTitle:**(NSString *)*aString*

<<forthcoming>>

synchronizeTitleAndSelectedItem

– (void)**synchronizeTitleAndSelectedItem**

Ensures that the item being displayed by the receiver agrees with the selected item (see **indexOfSelectedItem**). If there's no selected item, this method selects the first item in the item list and sets the receiver's item to match. For pull-down lists, this method makes sure that the first item is being displayed (the NSPopUpButtonCell must be set to use the selected menu item, which happens by default).

See also: – **itemArray**

titleOfSelectedItem

– (NSString *)**titleOfSelectedItem**

Returns the title of the item last selected by the user or the empty string if there's no such item.

See also: – **indexOfSelectedItem**

Notifications

NSPopUpButtonWillPopUpNotification

Posted when the NSPopUpButton receives a mouse-down event; that is, when the user is about to select an item from the list. This notification contains a notification object but no userInfo dictionary. The notification object is the selected NSPopUpButton.

NSPrinter

Inherits From: NSObject

Conforms To: NSCoding
NSCopying
NSObject (NSObject)

Declared In: AppKit/NSPrinter.h

Class Description

An NSPrinter object describes a printer's capabilities, such as whether the printer can print in color and whether it provides a particular font. An NSPrinter object represents either a particular make or type of printer, or an actual printer available to the computer. You use NSPrinter to get information about printers, not modify printer attributes or control a printing job.

There are two ways to create an NSPrinter:

- To create an abstract object that provides information about a type of printer rather than an object that represents an actual printer device, use the **printerWithType:** class method, passing a printer type (an NSString) as the argument. The **printerTypes** class method provides a list of printer types, model names recognized by the computer. Printer types are described in files written in PostScript Printer Description (PPD) format. The location of these files is platform dependent.
- Use the **printerWithName:** class method to create or find an NSPrinter that corresponds to an actual printer device. Use the **printerNames** class method to get a list of recognized printer names.

Once you have an NSPrinter, there's only one thing you can do with it: retrieve information regarding the type of printer or regarding the actual printer the object represents.

When you create an NSPrinter object, the object reads the file that corresponds to the type of printer, a model name, you specified and stores the data it finds there in named tables. Printer types are described in files written in the PostScript Printer Description (PPD) format. Any piece of information in the PPD tables can be retrieved through the methods **stringForKey:inTable:** and **stringListForKey:inTable:**, as explained later. Commonly needed items, such as whether a printer supports color or the size of the page on which it prints, are available through more direct methods (methods such as **isColor** and **pageSizeForPaper:**).

Note: To understand what the NSPrinter tables contain, you need to be acquainted with the PPD file format. This is described in *PostScript Printer Description File Format Specification, version 4.0*, available from Adobe Systems Incorporated. The rest of this class description assumes a familiarity with the concepts and terminology presented in the Adobe manual. A brief summary of the PPD format is given below; PPD terms defined in the Adobe manual are shown in italic.

PPD Format

A PPD file statement, or *entry*, associates a value with a main keyword:

**mainKeyword: value*

The asterisk is literal; it indicates the beginning of a new entry.

For example:

```
*ModelName: "MMimeo Machine"
*3dDevice: False
```

A main keyword can be qualified by an *option keyword*:

**mainKeyword optionKeyword: value*

For example:

```
*PaperDensity Letter: "0.1"
*PaperDensity Legal: "0.2"
*PaperDensity A4: "0.3"
*PaperDensity B5: "0.4"
```

In addition, any number of entries may have the same main keyword with no option keyword yet give different values:

```
*InkName: ProcessBlack/Process Black
*InkName: CustomColor/Custom Color
*InkName: ProcessCyan/Process Cyan
*InkName: ProcessMagenta/Process Magenta
*InkName: ProcessYellow/Process Yellow
```

Option keywords and values can sport *translation strings*. A translation string is a textual description, appropriate for display in a user interface, of the option or value. An option or value is separated from its translation string by a slash:

```
*Resolution 300dpi/300 dpi: " ... "
*InkName: ProcessBlack/Process Black
```

In the first example, the **300dpi** option would be presented in a user interface as “300 dpi.” In the second example, the translation string for the **ProcessBlack** value is set to “Process Black”.

NSPrinter treats entries that have an ***OrderDependency** or ***UIConstraint** main keyword specially. Such entries take the following forms (the bracketed elements are optional):

*OrderDependency: *real section mainKeyword [optionKeyword]*

*UIConstraint: *mainKeyword1 [optionKeyword1] mainKeyword2 [optionKeyword2]*

There may be more than one UIConstraint entry with the same *mainKeyword1* or *mainKeyword1/optionKeyword1* value. Below are some examples of ***OrderDependency** and ***UIConstraint** entries:

```
*OrderDependency: 10 AnySetup *Resolution
*UIConstraint: *Option3 None *PageSize Legal
*UIConstraint: *Option3 None *PageRegion Legal
```

Explaining these entries is beyond the scope of this documentation; however, it's important to note their forms in order to understand how they're represented in the NSPrinter tables.

NSPrinter Tables

NSPrinter defines five key-value tables to store PPD information. The tables are identified by the names given below:

Name	Contents
PPD	General information about a printer type. This table contains the values for all entries in a PPD file except those with the *OrderDependency and ***UIConstraint** main keywords. The values in this table don't include the translation strings.
PPDOptionTranslation	Option keyword translation strings.
PPDArgumentTranslation	Value translation strings.
PPDOrderDependency	***OrderDependency** values.
PPDUIConstraints	***UIConstraint** values.

There are two principle methods for retrieving data from the NSPrinter tables:

- **stringForKey:inTable:** returns the value for the first occurrence of a given key in the given table.
- **stringListForKey:inTable:** returns an array of values, one for each occurrence of the key.

For both methods, the first argument is an NSString that names a key—which part of a PPD file entry the key corresponds to depends on the table (as explained in the following sections). The second argument names the table that you want to search. The values that are returned by these methods, whether singular or in an array, are always NSStrings, even if the value wasn't a quoted string in the PPD file.

The NSPrinter tables store data as ASCII text, thus the two methods described above are sufficient for retrieving any value from any table. NSPrinter provides a number of other methods, such as **booleanForKey:inTable:** and **intForKey:inTable:**, that retrieve single values and coerce them, if possible, into particular data types. The coercion doesn't affect the data that's stored in the table (it remains in ASCII format).

To check the integrity of a table, use the **isKey:forTable:** and **statusForTable:** methods. The former returns a boolean that indicates whether the given key is valid for the given table; the latter returns an error code that describes the general state of a table (in particular, whether it actually exists).

Retrieving Values from the PPD Table

Keys for the PPD table are strings that name a main keyword or main keyword/option keyword pairing (formatted as *"mainKeyword/optionKeyword"*). In both cases, you exclude the main keyword asterisk. The following example creates an NSPrinter and invokes **stringForKey:inTable:** to retrieve the value for an un-optioned main keyword:

```
/* Create an NSPrinter object for a printer type. */
NSPrinter *prType = [NSPrinter printerWithType:@"My_Mimeo_Machine"]

/* Sets sValue to FALSE. */
NSString *sValue = [prType stringForKey:@"3dDevice" inTable:@"PPD"];
```

To retrieve the value for a main keyword/option keyword pair, pass the keywords formatted as *"mainKeyword/optionKeyword"*:

```
/* Sets sValue to "0.3". */
NSString *sValue = [prType stringForKey:@"PaperDensity/A4" inTable:@"PPD"];
```

stringForKey:inTable: can determine if a main keyword has options. If you pass a main keyword (only) as the first argument to the method, and if that keyword has options in the PPD file, the method returns an empty string. If it doesn't have options, it returns the value of the first occurrence of the main keyword:

```
/* Sets sValue to an empty string. */
NSString *sValue = [prType stringForKey:@"PaperDensity" inTable:@"PPD"];

/* Sets sValue to "ProcessBlack". */
NSString *sValue = [prType stringForKey:@"InkName" inTable:@"PPD"];
```

To retrieve the values for all occurrences of a main keyword, use the **stringListForKey:inTable:** method giving the main keyword only:

```
/* Sets sList to an array containing "ProcessBlack","CustomColor", etc. */
NSArray *sList = [prType stringListForKey:@"InkName" inTable:@"PPD"];
```

In addition, **stringListForKey:inTable:** can be used to retrieve all the options for a main keyword (given that the main keyword has options):

```
/* Sets sList to an array containing "Letter", "Legal","A4", etc. */
NSArray *sList = [prType stringListForKey:@"PaperDensity" inTable:@"PPD"];
```

Retrieving Values from the Option and Argument Translation Tables

A key to a translation table is similiar to a key to the PPD table: It's a main keyword or main/option keyword pair (again excluding the asterisk). However, the values that are returned from the translation tables are the translation strings for the option or argument (value) portions of the PPD file entry. For example:

```
/* Sets sValue to "300 dpi". */
NSString *sValue = [prType stringForKey:@"Resolution/300dpi"
    inTable:@"PPDOptionTranslation"];
```

```
/* Sets sList to an array containing "Process Black", "Custom Color", etc. */
NSArray *sList = [prType stringListForKey:@"InkName"
    inTable:@"PPDArgumentTranslation"];
```

As with the PPD table, use **stringListForKey:inTable:** to request an array of all occurances of a main keyword.

Retrieving Values from the Order Dependency Table

As mentioned earlier, an order dependency entry takes this form:

*OrderDependency: *real section mainKeyword [optionKeyword]*

These entries are stored in the PPDOrderDependency table. To retrieve a value from this table, always use **stringListForKey:inTable:**. The value passed as the key is, again, a main keyword or main keyword/option keyword pair; however, these values correspond to the *mainKeyword* and *optionKeyword* parts of an order dependency entry's value. As with the other tables, the main keyword's asterisk is excluded. The method returns an NSArray of two NSStrings that correspond to the *real* and *section* values for the entry. For example:

```
/* Sets sList to an array containing "10" and "AnySetup". */
NSArray *sList = [prType stringListForKey:@"Resolution"
    inTable:@"PPDOrderDependency"]
```

Retrieving Values from the UIConstraints Table

Retrieving a value from the PPDUIConstraints table is similar to retrieving a value from the PPDOrderDependency table: always use **stringListForKey:inTable:** and the key corresponds to elements in the entry's value. Given the following form (as described earlier), the key corresponds to *mainKeyword1/optionKeyword1*:

*UIConstraint: *mainKeyword1* [*optionKeyword1*] *mainKeyword2* [*optionKeyword2*]

The NSArray that's returned by **stringListForKey:inTable:** contains the *mainKeyword2* and *optionKeyword2* values (with the keywords stored as separate elements in the NSArray) for every ***UIConstraints** entry that has the given *mainKeyword1/optionKeyword1* value. For example:

```
/* Sets sList to an array containing:
    "PageSize", "Legal", "PageRegion", and "Legal" */
NSArray *sList = [prType stringListForKey:@"Option3/None"
    inTable:@"PPDUIConstraints"]
```

Note that the main keywords that are returned in the NSArray don't have asterisks. Also, the NSArray that's returned always alternates main and option keywords. If a particular main keyword doesn't have an option associated with it, the string for the option will be empty (but the entry in the NSArray for the option *will* exist).

Adopted Protocols

NSCoding

– encodeWithCoder:
– initWithCoder:

NSCopying

– copyWithZone:

Method Types

Creating an NSPrinter

+ printerWithName:
+ printerWithName:domain:includeUnavailable:
+ printerWithType:

Getting general printer information

+ printerNames
+ printerTypes

Getting attributes

– domain
– host
– name
– note
– type

Getting specific information

– acceptsBinary
– imageRectForPaper:
– pageSizeForPaper:
– isColor
– isFontAvailable:
– isOutputStackInReverseOrder
– languageLevel

Querying the tables

– isKey:inTable:
– stringForKey:inTable:
– stringListForKey:inTable:
– booleanForKey:inTable:
– floatForKey:inTable:
– intForKey:inTable:
– rectForKey:inTable:
– sizeForKey:inTable:
– statusForTable:
– deviceDescription

Class Methods

printerNames

+ (NSArray *)printerNames

Returns an array of recognized printer names.

See also: **+printerTypes, – name**

printerTypes

+ (NSArray *)**printerTypes**

Returns an array of recognized model names.

See also: +**printerNames, – type**

printerWithName:

+ (NSPrinter *)**printerWithName:**(NSString *)*name*

Returns an NSPrinter that represents an actual printer with the given *name*. Returns **nil** if the specified printer is not available.

See also: **+ printerWithType:**, + **printerNames**, – **name**

printerWithName:domain:includeUnavailable:

+ (NSPrinter *)**printerWithName:**(NSString *)*name*
domain:(NSString *)*domain*
includeUnavailable:(BOOL)*flag*

This method is for Mach platforms only—it is not defined for other platforms. Returns an NSPrinter that represents an actual printer with the given *name* and *domain*. If *domain* is **nil**, the first printer (matching *name*) found on any host or domain is used. Returns **nil** if the specified printer is not available and *flag* is NO. If *flag* is YES, the availability of the printer is ignored.

See also: **+ printerWithName:, + printerWithType:**, + **printerNames, – domain, – name**

printerWithType:

+ (NSPrinter *)**printerWithType:**(NSString *)*type*

Returns an NSPrinter with the given printer *type*.

See also: **+ printerWithName:**, + **printerTypes, – type**

Instance Methods

acceptsBinary

– (BOOL)**acceptsBinary**

Returns YES if the receiver accepts binary PostScript, otherwise NO.

booleanForKey:inTable:

– (BOOL)**booleanForKey:**(NSString *)*key* **inTable:**(NSString *)*table*

Returns a boolean value associated with *key* in *table*. Will also return NO if *key* is not in *table*.

See also: **– isKey:inTable:**, **– stringForKey:inTable:**

copyWithZone:

@protocol NSCopying
– (id)**copyWithZone:**(NSZone *)*zone*

Doesn't return a copy of the receiver. Returns the receiver with its reference count incremented (sends **retain** to the receiver).

deviceDescription

– (NSDictionary *)**deviceDescription**

Returns a dictionary of keys and values describing the device. See NSGraphics.h for possible keys.

domain

– (NSString *)**domain**

This method is for Mach platforms only—it is not defined for other platforms. Returns the name of the domain in which the receiver's printer resides. Returns **nil** if the receiver doesn't represent an actual printer.

See also: **+ printerWithName:domain:includeUnavailable:**

floatForKey:inTable:

– (float)**floatForKey:**(NSString *)*key* **inTable:**(NSString *)*table*

Returns a floating-point value associated with *key* in *table*. Returns 0.0 if *key* is not in *table*.

See also: **– isKey:inTable:**, **– stringForKey:inTable:**

host

– (NSString *)**host**

Returns the name of the receiver's host computer.

imageRectForPaper:

– (NSRect)**imageRectForPaper:**(NSString *)*paperName*

Returns the printing rectangle for the paper *paperName*. Possible values for *paperName* are contained in the printer's PPD file. Typical values are Letter and Legal.

See also: **– pageSizeForPaper:**

intForKey:inTable:

– (int)**intForKey:**(NSString *)*key* **inTable:**(NSString *)*table*

Returns an integer value associated with *key* in *table*. Returns 0 if *key* is not in *table*.

See also: **– isKey:inTable:**, **– stringForKey:inTable:**

isColor

– (BOOL)**isColor**

Returns YES if the receiver can print color, otherwise NO.

isFontAvailable:

– (BOOL)**isFontAvailable:**(NSString *)*faceName*

Returns YES if font *faceName* is available to the receiver, otherwise NO.

isKey:inTable:

– (BOOL)**isKey:**(NSString *)*key* **inTable:**(NSString *)*table*

Returns YES if *key* is in *table*, otherwise NO.

isOutputStackInReverseOrder

– (BOOL)**isOutputStackInReverseOrder**

Returns YES if the receiver outputs pages in reverse page order, otherwise NO.

languageLevel

– (int)**languageLevel**

Returns the PostScript Language Level recognized by the receiver.

name

– (NSString *)**name**

Returns the receiver's name.

See also: + **printerNames, + printerWithName:**

note

– (NSString *)**note**

Returns the note associated with the receiver.

pageSizeForPaper:

– (NSSize)**pageSizeForPaper:**(NSString *)*paperName*

Returns the size of the page for the paper type *paperName*. Possible values for *paperName* are contained in the printer's PPD file. Typical values are Letter and Legal.

See also: – **imageRectForPaper:**

rectForKey:inTable:

– (NSRect)**rectForKey:**(NSString *)*key* **inTable:**(NSString *)*table*

Returns the rectangle associated with *key* in *table*. Returns NSZeroRect if *key* is not in *table*.

See also: – **isKey:inTable:**, – **stringForKey:inTable:**

sizeForKey:inTable:

– (NSSize)**sizeForKey:**(NSString *)*key* **inTable:**(NSString *)*table*

Returns the size associated with *key* in *table*. The returned width and height is 0.0 if *key* is not in *table*.

See also: – **isKey:inTable:**, – **stringForKey:inTable:**

statusForTable:

– (NSPrinterTableStatus)**statusForTable:**(NSString *)*table*

Returns the status of *table*:

```
NSPrinterTableOK
NSPrinterTableNotFound
NSPrinterTableError
```

stringForKey:inTable:

– (NSString *)**stringForKey:**(NSString *)*key* **inTable:**(NSString *)*table*

Returns the first occurence of a value associated with *key* in *table*. If *key* is a main keyword only, and if that keyword has options in the PPD file, this method returns an empty string. Use **stringListForKey:inTable:** to retrieve the values for all occurrences of a main keyword. Returns **nil** if *key* is not in *table*.

See also: – **isKey:inTable:**, – **booleanForKey:inTable:**, – **floatForKey:inTable:**, – **intForKey:inTable:**, – **rectForKey:inTable:**, – **sizeForKey:inTable:**

stringListForKey:inTable:

– (NSArray *)**stringListForKey:**(NSString *)*key* **inTable:**(NSString *)*table*

Returns an array of strings, one for each occurrence, associated with *key* in *table*. Returns **nil** if *key* is not in *table*.

See also: **– isKey:inTable:**, **– stringForKey:inTable:**

type

– (NSString *)**type**

Returns the name of the receiver's type.

See also: + **printerTypes**

NSPrintInfo

Inherits From: NSObject

Conforms To: NSCoding
NSCopying
NSObject (NSObject)

Declared In: AppKit/NSPrintInfo.h

Class Description

An NSPrintInfo object stores information that's used to generate PostScript output. A shared NSPrintInfo object is automatically created for an application and is used by default for all printing jobs for that application. You can create any number of additional NSPrintInfo objects; however, only one can be "active" at a time, set using the **setSharedPrintInfo**: class method. You get the shared NSPrintInfo object using the **sharedPrintInfo** class method.

An NSPrintInfo object is used by NSPrintOperation objects to control printing (it is passed to a NSPrintOperation object which makes a copy of it to use during an operation). If you create special instances of NSPrintInfo objects for a specific printing task, you must ensure that your NSPrintInfo object is the shared one, or instantiate an NSPrintOperation object specifying your NSPrintInfo object.

Normally you don't set NSPrintInfo attributes directly—this is done by instances of NSPageLayout and NSPrintPanel. The NSView that's being printed may also supercede some NSPrintInfo settings. In particular, an NSView can supply the range of pages in the document and can provide its own pagination mechanism through the **knowsPagesFirst:last:** and **rect:forPage:** methods (see the documentation of these methods in the NSView class for details).

If the NSView doesn't supply pagination information, the NSPrintInfo's vertical and horizontal pagination constants are used to trigger these built-in pagination mechanisms:

Pagination Constant	Meaning
NSAutoPagination	The image is diced into equal-sized rectangles and placed in one column of pages.
NSFitPagination	The image is scaled to produce one column or one row of pages.
NSClipPagination	The image is clipped to produce one column or row of pages.

Vertical and horizontal pagination needn't be the same. However, if either dimension is scaled (NSFitPagination), the other dimension is scaled by the same amount to avoid stretching the image. If both dimensions are scaled, the scaling factor that produces the smallest image is used. Note that NSPrintInfo's scaling factor is independent of the scaling that's imposed by pagination and is applied after the document has been paginated.

NSPrintInfo uses points as the unit of measurement for paper size and margin width in the methods described below. See the NSFont specification for a discussion of points.

Adopted Protocols

NSCoding

– encodeWithCoder:
– initWithCoder:

NSCopying

– copyWithZone:

Method Types

Initializing an NSPrintInfo

– initWithDictionary:

Managing the shared NSPrintInfo

+ setSharedPrintInfo:
+ sharedPrintInfo

Managing the printing rectangle

+ sizeForPaperName:
– bottomMargin
– leftMargin
– orientation
– paperName
– paperSize
– rightMargin
– setBottomMargin:
– setLeftMargin:
– setOrientation:
– setPaperName:
– setPaperSize:
– setRightMargin:
– setTopMargin:
– topMargin

Pagination

– horizontalPagination
– setHorizontalPagination:
– setVerticalPagination:
– verticalPagination

Positioning the image on the page

– isHorizontallyCentered
– isVerticallyCentered
– setHorizontallyCentered:
– setVerticallyCentered:

Specifying the printer

+ defaultPrinter
+ setDefaultPrinter:
– printer
– setPrinter:

Controlling printing

– jobDisposition
– setJobDisposition:
– setUpPrintOperationDefaultValues

Accessing the dictionary

– dictionary

Class Methods

defaultPrinter

+ (NSPrinter *)**defaultPrinter**

Returns the user's default printer. Returns **nil** if the printer can not be found.

See also: + **setDefaultPrinter:**

setDefaultPrinter:

+ (void)**setDefaultPrinter:**(NSPrinter *)*aPrinter*

Sets the user's default printer to *aPrinter*. Unless the receiver's printer was specified using **setPrinter:**, this default printer is used.

See also: + **defaultPrinter**, – **printer**

setSharedPrintInfo:

+ (void)**setSharedPrintInfo:**(NSPrintInfo *)*printInfo*

Sets the shared NSPrintInfo object to *printInfo*. The shared NSPrintInfo object defines the settings for the NSPageLayout panel and print operations that will be used if no NSPrintInfo object is specified for those operataions. *printInfo* should never be **nil**.

See also: + **sharedPrintInfo**

sharedPrintInfo

+ (NSPrintInfo *)**sharedPrintInfo**

Returns the shared NSPrintInfo object.

See also: + **setSharedPrintInfo:**

sizeForPaperName:

+ (NSSize)**sizeForPaperName:**(NSString *)*name*

Returns the size for the specified type of paper in points. *name* identifies the type of paper, such as Letter or Legal. Paper names are implementation specific.

Instance Methods

bottomMargin

– (float)**bottomMargin**

Returns the height of the bottom margin in points.

See also: – **setBottomMargin:**

dictionary

– (NSMutableDictionary *)**dictionary**

Returns the receiver's dictionary that stores its attribute settings. The key/value pairs contained in the dictionary are described in **initWithDictionary:**. Note, modifying the returned dictionary will change the receiver's attributes.

horizontalPagination

– (NSPrintingPaginationMode)**horizontalPagination**

Returns the horizontal pagination mode, see **setHorizontalPagination:** for description of return values.

See also: **– setVerticalPagination:, – verticalPagination**

initWithDictionary:

– (id)**initWithDictionary:**(NSDictionary *)*aDictionary*

Initializes a newly allocated NSPrintInfo object by assigning it the parameters specified in *aDictionary*. This is the designated initializer for this class. The possible key/value pairs contained in *aDictionary* are listed below. Non-object values should be stored as NSValues in the dictionary.

Key	Type	Description
NSPrintPaperName	NSString	The paper name.
NSPrintPaperSize	NSSize	Height and width of paper in points.
NSPrintMustCollate	BOOL	If YES, collates output.
NSPrintFormName	NSString	Form name such as "Letter" or "Letter Small".
NSPrintOrientation	NSPrintingOrientation	NSPortraitOrientation or NSLandscapeOrientation
NSPrintLeftMargin	float	The left margin in points.
NSPrintRightmargin	float	The right margin in points.
NSPrintTopMargin	float	The top margin in points.
NSPrintBottomMargin	float	The bottom margin in points.
NSPrintHorizontallyCentered	BOOL	If YES, pages are centered horizontally.
NSPrintVerticallyCentered	BOOL	If YES, pages are centered vertically.
NSPrintHorizontalPagination	NSPrintingPaginationMode	NSAutoPagination, NSFitPagination, or NSClipPagination. See setHorizontalPagination: for details.
NSPrintVerticalPagination	NSPrintingPaginationMode	NSAutoPagination, NSFitPagination, or NSClipPagination. See setVerticalPagination: for details.
NSPrintScalingFactor	float	Scale factor before pagination.
NSPrintAllPages	BOOL	If YES, includes all pages in output.
NSPrintReversePageOrder	BOOL	If YES, prints last page first.
NSPrintFirstPage	int	The first page in the print job.

Key	Type	Description
NSPrintLastPage	int	The last page in the print job.
NSPrintCopies	int	Number of copies to spool.
NSPrintPagesPerSheet	int	The number of pages of the document that are printed on a single sheet of paper. This number is rounded up to the power of two when used by the system.
NSPrintJobFeatures	NSMutableDictionary	Features from the NSPrinter object where keys are the feature name and values are the settings. For example, the key NSPrintPaperFeed might have the value *InputSlot/Upper. See the NSPrinter class description.
NSPrintPaperFeed	NSString	The printer slot. For example, InputSlot/Upper or NSPrintManualFeed.
NSPrintPrinter	NSPrinter	The printer to use.
NSPrintJobDisposition	NSString	NSPrintSpoolJob, NSPrintFaxJob, NSPrintPreviewJob, NSPrintSaveJob, or NSPrintCancelJob. See setJobDisposition: for details.
NSPrintSavePath	NSString	Pathname to save as a file if job disposition is NSPrintSaveJob.
NSPrintFaxReceiverNames	NSArray	Array of NSStrings containing receiver names for a fax job.
NSPrintFaxReceiverNumbers	NSArray	Array of NSStrings containing the receiver phone numbers for a fax job.
NSPrintFaxSendTime	NSDate	When to send the fax.
NSPrintFaxUseCoverSheet	BOOL	If YES, send cover sheet.
NSPrintFaxCoverSheetName	NSString	The filename containing the cover sheet.
NSPrintFaxReturnReceipt	BOOL	If YES, sends confirmation email when fax is sent.
NSPrintFaxHighResolution	BOOL	If YES, sends fax at high resolution.

Key	Type	Description
NSPrintFaxTrimPageEnds	BOOL	If YES, trims page ends, otherwise sends complete pages.
NSPrintFaxModem	NSPrinter	The fax modem to use.

See also: – **dictionary**

isHorizontallyCentered

– (BOOL)**isHorizontallyCentered**

Returns YES if the image is centered horizontally, otherwise returns NO.

See also: – **isVerticallyCentered**, – **setHorizontallyCentered:**

isVerticallyCentered

– (BOOL)**isVerticallyCentered**

Returns YES if the image is centered vertically, otherwise returns NO.

See also: – **isHorizontallyCentered**, – **setVerticallyCentered:**

jobDisposition

– (NSString *)**jobDisposition**

Returns the action specified for the job. See **setJobDisposition:** for description of return values.

leftMargin

– (float)**leftMargin**

Returns the width of the left margin in points.

See also: – **setLeftMargin:**

orientation

– (NSPrintingOrientation)**orientation**

Returns the orientation attribute. See **setOrientation:** for description of return values.

paperName

– (NSString *)**paperName**

Returns the paper name such as "Letter" or "Legal". Paper names are implementation specific.

See also: – **setPaperName:**

paperSize

– (NSSize)**paperSize**

Returns the size of the paper in points.

See also: – **setPaperSize:**

printer

– (NSPrinter *)**printer**

Returns the NSPrinter to be used for printing.

See also: – **setPrinter:**

rightMargin

– (float)**rightMargin**

Returns the width of the right margin in points.

See also: – **setRightMargin:**

setBottomMargin:

– (void)**setBottomMargin:**(float)*margin*

Sets the bottom margin to *margin* specified in points.

See also: – **bottomMargin**

setHorizontalPagination:

– (void)**setHorizontalPagination:**(NSPrintingPaginationMode)*mode*

Sets the horizontal pagination to *mode* where *mode* is one of:

Mode	Meaning
NSAutoPagination	The image is diced into equal-sized rectangles and placed in one column of pages.
NSFitPagination	The image is scaled to produce one column or one row of pages.
NSClipPagination	The image is clipped to produce one column or row of pages.

See also: – **horizontalPagination**, – **setVerticalPagination:, – verticalPagination**

setHorizontallyCentered:

– (void)**setHorizontallyCentered:**(BOOL)*flag*

If *flag* is YES the image will be centered horizontally.

See also: – **isHorizontallyCentered, – isVerticallyCentered, – setVerticallyCentered:**

setJobDisposition:

– (void)**setJobDisposition:**(NSString *)*disposition*

Sets the action specified for the job to *disposition*, where *disposition* is one of:

Disposition	Meaning
NSPrintSpoolJob	Normal print job.
NSPrintFaxJob	Fax job.
NSPrintPreviewJob	Send to Preview application.
NSPrintSaveJob	Save to a file.
NSPrintCancelJob	Cancel print job.

See also: – **jobDisposition**

setLeftMargin:

– (void)**setLeftMargin:**(float)*margin*

Sets the left margin to *margin* specified in points.

See also: – **leftMargin**

setOrientation:

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

Sets the page orientation to *orientation* where *orientation* is either NSPortraitOrientation or NSLandscapeOrientation. This method may change either the paper name or size for consistency. To avoid this side effect set the values in the dictionary directly.

See also: – **dictionary**, – **initWithDictionary:**, – **orientation**

setPaperName:

– (void)**setPaperName:**(NSString *)*name*

Sets the paper name to *name* (i.e., Letter or Legal). Paper names are implementation specific.This method may change either the size or orientation for consistency. To avoid this side effect set the values in the dictionary directly.

See also: – **dictionary**, – **initWithDictionary:**, – **paperName**

setPaperSize:

– (void)**setPaperSize:**(NSSize)*aSize*

Sets the width and height of the paper to *aSize* specified in points. This method may change either the paper name or orientation for consistency. To avoid this side effect set the values in the dictionary directly.

See also: – **dictionary**, – **initWithDictionary:**, – **paperSize**

setPrinter:

– (void)**setPrinter:**(NSPrinter *)*aPrinter*

Sets the printer used in subsequent printing jobs to *aPrinter*. This method iterates through the dictionary. If a feature in the dictionary is not supported by this new printer (this is determined by a query to the PPD file), then that feature is removed from the dictionary

See also: – **printer**

setRightMargin:

– (void)**setRightMargin:**(float)*margin*

Sets the right margin to *margin* specified in points.

See also: – **rightMargin**

setTopMargin:

– (void)**setTopMargin:**(float)*margin*

Sets the top margin to *margin* specified in points.

See also: – **topMargin**

setUpPrintOperationDefaultValues

– (void)**setUpPrintOperationDefaultValues**

Invoked when the print operation is about to start. Subclasses may override this method to set default values for any attributes that are not set.

setVerticalPagination:

– (void)**setVerticalPagination:**(NSPrintingPaginationMode)*mode*

Sets the vertical pagination to *mode* where *mode* is one of:

Mode	Meaning
NSAutoPagination	The image is diced into equal-sized rectangles and placed in one column of pages.
NSFitPagination	The image is scaled to produce one column or one row of pages.
NSClipPagination	The image is clipped to produce one column or row of pages.

See also: – **horizontalPagination**, – **setHorizontalPagination:**, – **verticalPagination**

setVerticallyCentered:

– (void)**setVerticallyCentered:**(BOOL)*flag*

If *flag* is YES, the image will be vertically centered.

See also: – **isHorizontallyCentered**, – **isVerticallyCentered**, – **setHorizontallyCentered:**

topMargin

– (float)**topMargin**

Returns the top margin in points.

See also: – **setTopMargin:**

verticalPagination

– (NSPrintingPaginationMode)**verticalPagination**

Returns the vertical pagination mode, see **setVerticalPagination:** for description of return values.

See also: – **horizontalPagination, – setHorizontalPagination:**

NSPrintOperation

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: AppKit/NSPrintOperation.h

Class Description

An NSPrintOperation object controls operations that generate Encapsulated PostScript (EPS) code or PostScript print jobs. Generally, EPS code is used to transfer images between applications, which happens when the user copies and pastes graphics, uses a Service, or uses ObjectLinks. PostScript is generated when the user prints and faxes documents. An NSPrintOperation object does not generate PostScript code itself; it just controls the overall process, relying on an NSView object to generate the actual code.

NSPrintOperation works in conjunction with two other objects: an NSPrintInfo object, which specifies how the code should be generated, and an NSView object, which performs the actual code generation. You specify these two objects in the method you use to create an NSPrintOperation object. If no NSPrintInfo is specified, NSPrintOperation uses the shared NSPrintInfo, which contains default values. (A shared NSPrintInfo object is automatically created for an application.) The shared NSPrintInfo works well for applications that are not document-based. However, document-based applications should create an NSPrintInfo for each document that might be printed or copied and use that object instead. This will allow users to set printing attributes on a per-document basis.

You create NSPrintOperation objects in any method that is invoked when a user chooses a Print or Copy command. That method must also send **runOperation** to the NSPrintOperation object to start the actual operation. For example, applications that are not document-based have a simple **print:** method as in:

```
- (void)print:sender {
    [[NSPrintOperation printOperatationWithView:self] runOperation];
}
```

However, document-based applications should use their own instances of NSPrintInfo as in:

```
- (void)print:sender {
    [[NSPrintOperation printOperationWithView:[self myView] printInfo:[document
        docPrintInfo]] runOperation];
}
```

This method creates an NSPrintOperation for a print job that uses the document’s NSPrintInfo object—not the shared NSPrintInfo object.

In both examples, because this is a print job, the NSPrintOperation object will display an NSPrintPanel object allowing the user to select printing options (i.e., number of pages to print and range of pages to print).

The NSPrintOperation object copies the NSPrintInfo object, updates this copy with information from the NSPrintPanel object, and uses the specified NSView to perform the operation. Some of the information stored in an NSPrintInfo object is constant for a particular document, such as its page size. Other information that is likely to change between print operations is set to default values before the operation begins. In this way, even though NSPrintOperation updates the NSPrintInfo with information from the NSPrintPanel for a specific print job, that information is reset back to the default values for each print job. Because NSPrintOperation keeps a copy of the NSPrintInfo it uses, you could duplicate a specific print job by storing and reusing that copy.

When repeating a print job, you can suppress the display of the NSPrintPanel object by sending **setShowPanels:**, passing NO as the argument, to the NSPrintOperation object before sending it **runOperation**. However, make sure that any non-default settings in the NSPrintInfo object that would normally be selected from a NSPrintPanel object are set to reasonable values—a copy of an NSPrintInfo object used in a previous print job will have the correct values.

You can also customize the NSPrintPanel object using the **setAccessoryView:** method or specify your own NSPrintPanel object using **setPrintPanel:** (an OpenStep addition).

If you want the PostScript code generated for EPS and printing to be the same but different from the code generated for the screen, you can test for this case by sending **isDrawingToScreen** to the current NSDPSContext as in:

```
if (![[NSDPSContext currentContext] isDrawingToScreen]){
    /* Insert EPS and printing code here */
}
```

If you want to generate different PostScript code when printing vs. creating EPS, you can test for this case by sending **isEPSOperation** to the current operation as follows:

```
if ((![[NSDPSContext currentContext] isDrawingToScreen] &&
        ![[NSPrintOperation currentOperation] isEPSOperation])){
    /* Insert printing code here */
}
```

Method Types

Creating an NSPrintOperation

+ EPSOperationWithView:insideRect:toData:
+ EPSOperationWithView:insideRect:toData:printInfo:
+ EPSOperationWithView:insideRect:toPath:printInfo:
+ printOperationWithView:
+ printOperationWithView:printInfo:
– initEPSOperationWithView:insideRect:toData:printInfo:
– initWithView:printInfo:

Setting the current NSPrintOperation for this thread

+ currentOperation
+ setCurrentOperation:

Determining the type of operation

– isEPSOperation

Modifying the NSPrintInfo object

– printInfo
– setPrintInfo:

Getting the NSView object

– view

Running a print operation

– runOperation
– cleanUpOperation
– deliverResult

Modifying the user interface

– showPanels
– setShowPanels:
– accessoryView
– setAccessoryView:
– printPanel
– setPrintPanel:

Managing the DPS context

– context
– createContext
– destroyContext

Modifying page information

– currentPage
– pageOrder
– setPageOrder:

Class Methods

EPSOperationWithView:insideRect:toData:

+ (NSPrintOperation *)**EPSOperationWithView:**(NSView *)*aView*
insideRect:(NSRect)*rect*
toData:(NSMutableData *)*data*

Returns a new NSPrintOperation object that controls the copying of EPS graphics from the area specified by *rect* in *aView*. The new NSPrintOperation object will use the default NSPrintInfo object. The EPS code

is written to *data*. Raises an NSPrintOperationExistsException if there is already a print operation in progress, otherwise the returned object is made the current print operation for this thread.

See also: + **EPSOperationWithView:insideRect:toData:printInfo:**, + **EPSOperationWithView:insideRect:toPath:printInfo:**

EPSOperationWithView:insideRect:toData:printInfo:

+ (NSPrintOperation *)**EPSOperationWithView:**(NSView *)*aView*
insideRect:(NSRect)*rect*
toData:(NSMutableData *)*data*
printInfo:(NSPrintInfo *)*aPrintInfo*

Returns a new NSPrintOperation object that controls the copying of EPS graphics from the area specified by *rect* in *aView*. The new NSPrintOperation object will use the settings stored in *aPrintInfo*. The code is written to *data*. Raises an NSPrintOperationExistsException if there is already a print operation in progress, otherwise the returned object is made the current print operation for this thread.

See also: + **EPSOperationWithView:insideRect:toData:**, + **EPSOperationWithView:insideRect:toPath:printInfo:**

EPSOperationWithView:insideRect:toPath:printInfo:

+ (NSPrintOperation *)**EPSOperationWithView:**(NSView *)*aView*
insideRect:(NSRect)*rect*
toPath:(NSString *)*path*
printInfo:(NSPrintInfo *)*aPrintInfo*

Creates and returns a new NSPrintOperation object that controls the copying of EPS graphics from the area specified by *rect* in *aView*. The new NSPrintOperation object will use the settings stored in *aPrintInfo*. The code is written to *path*. Raises an NSPrintOperationExistsException if there is already a print operation in progress, otherwise the returned object is made the current print operation for this thread.

See also: + **EPSOperationWithView:insideRect:toData:**, + **EPSOperationWithView:insideRect:toData:printInfo:**

currentOperation

+ (NSPrintOperation *)**currentOperation**

Returns the current print operation for this thread. Returns **nil** if there isn't a current operation.

See also: + **setCurrentOperation:**

printOperationWithView:

+ (NSPrintOperation *)**printOperationWithView:**(NSView *)*aView*

Returns a new NSPrintOperation that controls the printing of *aView*. The new NSPrintOperation object will use the settings stored in the shared NSPrintInfo object. Raises an NSPrintOperationExistsException if there is already a print operation in progress, otherwise the returned object is made the current print operation for this thread.

See also: + **printOperationWithView:printInfo:**

printOperationWithView:printInfo:

+ (NSPrintOperation *)**printOperationWithView:**(NSView *)*aView*
printInfo:(NSPrintInfo *)*aPrintInfo*

Returns a new NSPrintOperation that controls the printing of *aView*. The new NSPrintOperation object will use the settings stored in *aPrintInfo*. Raises an NSPrintOperationExistsException if there is already a print operation in progress, otherwise the returned object is made the current print operation for this thread.

See also: + **printOperationWithView:**

setCurrentOperation:

+ (void)**setCurrentOperation:**(NSPrintOperation *)*operation*

Sets the current print operation for this thread to *operation*. If operation is **nil**, then there is no current print operation.

See also: + **currentOperation**

Instance Methods

accessoryView

– (NSView *)accessoryView

Returns the accessory view used by the NSPrintPanel object. You use **setAccessoryView:** to customize the default NSPrintPanel object without having to subclass NSPrintPanel or specify your own NSPrintPanel object.

See also: – **printPanel**, – **setPrintPanel:**, – **setShowPanels:**, – **showPanels**

cleanUpOperation

– (void)**cleanUpOperation**

Invoked by **runOperation** at the end of an operation to remove the receiver as the current operation. You typically do not invoke this method directly.

context

– (NSDPSContext *)**context**

Returns the receiver's DPS context used for generating output.

See also: – **createContext**, – **destroyContext**

createContext

– (NSDPSContext *)**createContext**

Creates the DPS context for output generation, using the receiver's NSPrintInfo settings. Do not invoke this method directly—it's invoked before any output is generated.

See also: – **context**, – **destroyContext**

currentPage

– (int)**currentPage**

Returns the page number of the page that is currently being printed.

See also: – **pageOrder**, – **setPageOrder:**

deliverResult

– (BOOL)**deliverResult**

Delivers the results generated by **runOperation** to the intended destination (i.e., the printer spool, or preview application). Returns YES if the operation was successful, otherwise NO. Do not invoke this method directly—it's invoked automatically when the operation is done generating the output.

destroyContext

– (void)**destroyContext**

Destroys the receiver's DPS context. Do not invoke this method directly—it's invoked at the end of a print operation.

See also: – **context**, – **createContext**

initEPSOperationWithView:insideRect:toData:printInfo:

– (id)**initEPSOperationWithView:**(NSView *)*aView*
insideRect:(NSRect)*rect*
toData:(NSMutableData *)*data*
printInfo:(NSPrintInfo *)*aPrintInfo*

Initializes and returns a newly allocated NSPrintOperation object to control the copying of EPS graphics from the area specified by *rect* in *aView*, using the settings stored in *aPrintInfo*. This method makes a copy of *aPrintInfo* —*aPrintInfo* is not used in the actual operation. The EPS code is written to *data*.

See also: – **initWithView:printInfo:**

initWithView:printInfo:

– (id)**initWithView:**(NSView *)*aView* **printInfo:**(NSPrintInfo *)*aPrintInfo*

Initializes and returns a newly allocated NSPrintOperation object to control the printing of *aView*, using the settings stored in *aPrintInfo*. This method makes a copy of *aPrintInfo* —*aPrintInfo* is not used in the actual operation. This method is the designated initializer for this class.

See also: – **initEPSOperationWithView:insideRect:toData:printInfo:**

isEPSOperation

– (BOOL)**isEPSOperation**

Returns YES if the receiver controls an EPS operation (initiated by a copy command), and NO if the receiver controls a printing operation (initiated by a print command).

pageOrder

– (NSPrintingPageOrder)**pageOrder**

Returns the order in which pages will be printed. See **setPageOrder:** for possible return values.

See also: – **currentPage**

printInfo

– (NSPrintInfo *)**printInfo**

Returns the receiver's NSPrintInfo object.

See also: – **setPrintInfo:**

printPanel

– (NSPrintPanel *)**printPanel**

Returns the NSPrintPanel object used when running the operation.

See also: – **accessoryView**, – **setAccessoryView**:, – **setPrintPanel:, – setShowPanels:, – showPanels**

runOperation

– (BOOL)**runOperation**

Runs the operation (i.e., copys an EPS graphic or prints a job). Returns YES if successful, otherwise NO.

See also: – **cleanUpOperation**, – **deliverResult**

setAccessoryView:

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

Allows you to augment the NSPrintPanel object by adding a custom NSView (by using this method you do not need to subclass NSPrintPanel or specify your own NSPrintPanel object). The NSPrintPanel is automatically resized to accommodate the new accessory view *aView*.

See also: – **accessoryView**, – **printPanel**, – **setPrintPanel:, – setShowPanels:, – showPanels**

setPageOrder:

– (void)**setPageOrder:**(NSPrintingPageOrder)*order*

Sets the order in which pages will be printed to *order* where *order* is one of:

Order	Meaning
NSAscendingPageOrder	Ascending (back to front) page order.
NSDescendingPageOrder	Descending (front to back) page order.
NSSpecialPageOrder	The spooler will not rearrange pages—they are print in the order received by the spooler.
NSUnknownPageOrder	No page order specified.

See also: – **currentPage**, – **pageOrder**

setPrintInfo:

– (void)**setPrintInfo:**(NSPrintInfo *)*aPrintInfo*

Sets the receiver's NSPrintInfo object to *aPrintInfo*.

See also: – **printInfo**

setPrintPanel:

– (void)**setPrintPanel:**(NSPrintPanel *)*panel*

Sets the receiver's NSPrintPanel used in the operation to *panel*.

See also: – **accessoryView**, – **printPanel**, – **setAccessoryView:**, **– setShowPanels:**, **– showPanels**

setShowPanels:

– (void)**setShowPanels:**(BOOL)*flag*

If *flag* is YES then the NSPrintPanel will be used in the operation, otherwise it will not.

See also: – **accessoryView**, – **printPanel**, – **setAccessoryView:**, **– setPrintPanel:**, **– showPanels**

showPanels

– (BOOL)**showPanels**

Returns YES if the NSPrintPanel will be used in the operation, otherwise NO.

See also: – **accessoryView**, – **printPanel**, – **setAccessoryView:**, **– setPrintPanel:, – setShowPanels:**

view

– (NSView *)**view**

Returns the NSView object that generates the actual EPS or PostScript code controlled by the receiver.

NSPrintPanel

Inherits From: NSPanel : NSObject

Conforms To: NSObject (NSObject)

Declared In: AppKit/NSPrintPanel.h

Note: On Mach platforms, NSPrintPanel inherits from NSPanel and conforms to NSCoding.

Class Description

NSPrintPanel creates a Print panel used to query the user for information about a print job, such as which pages to print and how many copies, and execute the Print command.

When a **print:** message is sent to an NSView or NSWindow, an NSPrintOperation object is created to control the print operation (see the NSPrintOperation class description for details). By default an NSPrintOperation object uses an NSPrintPanel unless it is sent the **setShowPanels:** message passing NO as the argument. Also, if you subclass NSPrintPanel, send the **setPrintPanel:** message to the NSPrintOperation object passing an instance of your subclass to ensure that it is used as the Print panel for that operation.

However, you rarely need to subclass NSPrintPanel since you can augment its display by adding a custom NSView using the **setAccessoryView:** method. The accessory view is displayed when the user clicks the Options button. (On Mach platforms, the panel is resized to accommodate the NSView that you add.) Note, however, that you don't have to create controls for special printer features. If a printer includes features in the "OpenUI" field of its PostScript Printer Description (PPD) table, these features will appear in the panel. For more information on a printer's PPD table, see the NSPrinter class description.

Typically, you get an NSPrintPanel by invoking the **printPanel** class method. When the class receives a **printPanel** message, it tries to reuse an existing panel rather than create a new one. When a panel is reused, its attributes are reset to the default values so that the effect is the same as returning a new panel. Because a Print panel may be reused, you shouldn't modify the instance returned by **printPanel**, except through the methods listed below. For example, you can set the accessory view, but not the arrangement of the buttons within the panel. If you must modify the Print panel substantially, create and manage your own instance using the **alloc...** and **init...** methods rather than the **printPanel** method.

An application stores printing information in an NSPrintInfo object. When an NSPrintOperation object is created it is given a specific NSPrintInfo object from the application or assigned a default. You can get the current operation by sending the **currentOperation** class method to NSPrintOperation.

Use the **updateFromPrintInfo** method to read the NSPrintInfo object's information into the Print panel. Conversely, the **finalWritePrintInfo** method updates the NSPrintInfo object if the user changes the

information on the Print panel. The NSPrintOperation object creates a copy of the NSPrintInfo object, so that **finalWritePrintInfo** actually writes to that copy, not the original.

Method Types

Creating an NSPrintPanel

+ printPanel

Customizing the panel

– accessoryView
– setAccessoryView:

Running the panel

– runModal

Communicating with the NSPrintInfo object

– updateFromPrintInfo
– finalWritePrintInfo

Updating the panel's display

– pickedButton:
– pickedAllPages:
– pickedLayoutList:

Class Methods

printPanel

+ (NSPrintPanel *)**printPanel**

Returns a shared NSPrintPanel object or a newly created one if it doesn't already exist.

Instance Methods

accessoryView

– (NSView *)**accessoryView**

Returns the receiver's accessory view (used to customize the receiver).

See also: – **setAccessoryView:**

finalWritePrintInfo

– (void)**finalWritePrintInfo**

Writes the values of the receiver's printing attributes to the NSPrintInfo object belonging to the current NSPrintOperation.

See also: – **updateFromPrintInfo**, **+ currentOperation** (NSPrintOperation)

pickedAllPages:

– (void)**pickedAllPages:**(id)*sender*

This method is for Mach platforms only—it is not defined for other platforms. Invoked when the user chooses the All (Pages) radio button. If the All button is clicked the From and To (Pages) fields are empty, otherwise their default values are set to "first" and "last". Override this method to change these defaults.

See also: – **pickedButton:**, – **pickedLayoutList:**

pickedButton:

– (void)**pickedButton:**(id)*sender*

This method is for Mach platforms only—it is not defined for other platforms. Invoked when the user clicks either the Cancel, Fax, Preview, Print or Save buttons. If a button other than the Cancel button is clicked, then the receiver's Copies, From (Pages) and To (Pages) fields must contain acceptable values (positive numbers), otherwise the unacceptable entry is selected. If the fields are acceptable the modal loop is stopped. If the Cancel button is selected the modal loop is stopped regardless of the field values.

See also: – **pickedAllPages:**, – **pickedLayoutList:**, – **runModal**

pickedLayoutList:

– (void)**pickedLayoutList:**(id)*sender*

This method is for Mach platforms only—it is not defined for other platforms. Invoked when the user chooses a new layout to update the receiver.

See also: – **pickedAllPages:**, – **pickedButton:**

runModal

– (int)**runModal**

Displays the receiver and begins the modal loop. Returns NSCancelButton if the user clicks the Cancel button, otherwise returns NSOkButton.

See also: – **pickedButton:**

setAccessoryView:

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

Adds an NSView to the receiver. Invoke this method to add a custom view containing your controls. The accessory view is displayed when the user clicks the Options button. (On Mach platforms, the receiver is automatically resized to accommodate *aView*.) This method can be invoked repeatedly to change the accessory view depending on the situation. If *aView* is **nil**, then the receiver's current accessory view, if any, is removed.

See also: – **accessoryView**

updateFromPrintInfo

– (void)**updateFromPrintInfo**

Reads the receiver's values from the NSPrintInfo object belonging to the current NSPrintOperation, and updates the receiver accordingly.

See also: – **finalWritePrintInfo**, + **currentOperation** (NSPrintOperation)

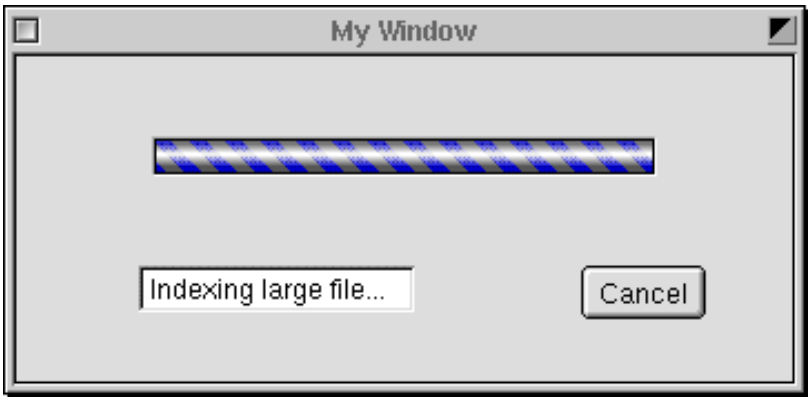
NSProgressIndicator

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

Class Description

An application displays a progress indicator to show that a lengthy task is under way. Some progress indicators do nothing more than spin to show that the application is busy, while others show the percentage of the task that has been completed. NSProgressIndicator provides both types of display:

- an indeterminate progress indicator (the “barber pole”) that spins until the task is complete (see illustration below)
- a determinate progress indicator that draws a three-dimensional progress bar from left to right in the view as the task progresses



NSProgressIndicator is a subclass of NSView. To display a progress indicator, your application creates a window and adds the progress indicator as a subview of the window’s content view or any subview. You can create a progress indicator programmatically and initialize it with the **initWithFrame:** method. However, you normally use Interface Builder to create and initialize a progress indicator and to install it in an application view.

For an indeterminate progress indicator, you invoke **startAnimation:** to start the animation (the spinning of the barber pole) and **stopAnimation:** when the task is complete. By default, the delay between animation

steps is one twelfth of a second (5.0/60.0). You can change the animation delay by invoking **setAnimationDelay:**. Setting the delay to a double value larger than the default value will slow the animation, while setting the delay to a smaller value will speed it up.

Instead of invoking **startAnimation:** and **stopAnimation:**, you can control an indeterminate progress indicator directly by sending the **animate:** message. Each time you invoke **animate:**, the animation advances by one step. You can speed up or slow down the animation by varying how often you invoke **animate:**. Like other views, a progress indicator redisplays itself on each pass through the event loop, if needed. To ensure immediate redrawing, however, you can invoke the **displayIfNeeded** method (inherited from NSView) each time you invoke **animate:**.

By default, a progress indicator is indeterminate. Your can specify a determinate progress indicator when you set up the view with Interface Builder, or you can use code like the following to change the default value programmatically:

```
[myProgressIndicatorView setIndeterminate:FALSE];
```

For a determinate progress indicator, you invoke the **incrementBy:** method to advance the progress bar. By default, a determinate progress indicator goes from 0.0 to 100.0. You can increment by any amount, but if you vary the increment too widely, progress may appear uneven or jerky. You typically choose an increment value that evenly divides 100.0. For example, you might invoke **incrementBy:** 50 times, incrementing by 2.0 each time, to draw the complete progress bar. To modify the default range of 0.0 to 100.0, you can invoke **setMinValue:** to modify the minimum value and **setMaxValue:** to modify the maximum value.

After each invocation of **incrementBy:**, you can invoke the **displayIfNeeded** method to ensure immediate redrawing.

You can display progress indicators of different sizes by varying the frame size. However, the default size is designed to provide the best results. By default, a progress indicator is drawn with a bezeled frame, but you can use the **setBezeled:** method to modify the bezeled-frame setting.

A progress indicator is drawn with colors based on the user's current color scheme. When the user changes the color scheme, the color of the progress indicator changes automatically to match the new scheme.

Method Types

Creating an instance

– initWithFrame:

Animating the progress indicator

– animate:
– animationDelay
– setAnimationDelay:
– setUsesThreadedAnimation:
– startAnimation:
– stopAnimation:
– usesThreadedAnimation

Advancing the progress bar

– incrementBy:
– setDoubleValue:
– doubleValue
– setMinValue:
– minValue
– setMaxValue:
– maxValue

Setting the appearance

– setBezeled:
– isBezeled
– setIndeterminate:
– isIndeterminate

Instance Methods

animate:

– (void)**animate:**(id)*sender*

For an indeterminate progress indicator, advances the progress animation by one step. For a determinate progress indicator, does nothing. Your application uses this method to control animation directly (as opposed to invoking **startAnimation:** and **stopAnimation:** for automatic animation). The more often you invoke **animate:**, the faster the animation progresses.

The xx method only invalidates the progress indicator so it will be redrawn the next time through the event loop. To ensure immediate redrawing, invoke the **displayIfNeeded** method.

See also: – **animationDelay**, – **setAnimationDelay:**

animationDelay

– (NSTimeInterval)**animationDelay**

For an indeterminate progress indicator, returns the delay, in seconds, between animation steps. By default, the animation delay is set to one twelfth of a second (5.0/60.0). A determinate progress indicator does not use the animation delay value.

See also: – **animate:**

doubleValue

– (double)**doubleValue**

For a determinate progress indicator, returns a value that indicates the current extent of the progress bar. For example, a determinate progress indicator goes from 0.0 to 100.0 by default. If the progress bar has advanced half way across the view, the value returned by **doubleValue** would be 50.0. An indeterminate progress indicator does not use this value.

See also: – **incrementBy:**, – **setDoubleValue:**

incrementBy:

– (void)**incrementBy:**(double)*delta*

For a determinate progress indicator, you invoke **incrementBy:** to advance the progress bar by *delta*. For example, if you want to advance a progress bar from 0.0 to 100.0 in twenty steps, you would invoke **incrementBy:** twenty times with a *delta* value of 5.0.

See also: – **doubleValue**

initWithFrame:

– (id)**initWithFrame:**(NSRect)*frameRect*

Initializes a newly allocated NSProgressIndicator with *frameRect* as its frame rectangle. This method is the designated initializer for the NSProgressIndicator class. It calls the **initWithFrame:** method of its superclass, NSView, then performs initialization specific to the outline view. It returns **self**.

It's usually more convenient to use Interface Builder, which allows you to create an NSProgressIndicator and embed it in the superview of your choice.

isBezeled

– (BOOL)**isBezeled**

Returns YES if the NSProgressIndicator's frame has a three-dimensional bezel.

See also: – **setBezeled:**

isIndeterminate

– (BOOL)**isIndeterminate**

Returns YES if the NSProgressIndicator is indeterminate. An indeterminate progress indicator displays a "barber pole" that spins until the task is complete. A determinate progress indicator draws a three-dimensional bar from left to right as the task progresses.

See also: – **setIndeterminate:**

maxValue

– (double)**maxValue**

For a determinate progress indicator, returns the maximum value for the progress bar. By default, a determinate progress indicator goes from 0.0 to 100.0, so the value returned would be 100.0. An indeterminate progress indicator does not use this value.

See also: – **minValue**, – **setMaxValue:**

minValue

– (double)**minValue**

For a determinate progress indicator, returns the minimum value for the progress bar. By default, a determinate progress indicator goes from 0.0 to 100.0, so the value returned would be 0.0. An indeterminate progress indicator does not use this value.

See also: – **maxValue**, – **setMinValue:**

setAnimationDelay:

– (void)**setAnimationDelay:**(NSTimeInterval)*delay*

Sets the delay, in seconds, between animation steps for an indeterminate progress indicator. By default, the animation delay is set to one twelfth of a second (5.0/60.0). Setting the delay to a double value larger than 5.0/60.0 will slow the animation, while setting the delay to a smaller value will speed it up. A determinate progress indicator does not use the animation delay value.

setBezeled:

– (void)**setBezeled:**(BOOL)*flag*

Sets whether the NSProgressIndicator's frame has a three-dimensional bezel.

See also: – **isBezeled**

setDoubleValue:

– (void)**setDoubleValue:**(double)*doubleValue*

Sets the value that indicates the current extent of the progress bar. An indeterminate progress indicator does not use this value.

See also: – **doubleValue**, – **incrementBy:**, – **setMaxValue:**, – **setMinValue:**

setIndeterminate:

– (void)**setIndeterminate**:(BOOL)*flag*

Sets whether the NSProgressIndicator is indeterminate.

See also: – **isIndeterminate**

setMaxValue:

– (void)**setMaxValue:**(double)*newMaximum*

Specifies the maximum value for the progress bar. An indeterminate progress indicator does not use this value.

See also: – **maxValue**

setMinValue:

– (void)**setMinValue:**(double)*newMinimum*

Specifies the minimum value for the progress bar. An indeterminate progress indicator does not use this value.

See also: – **minValue**

setUsesThreadedAnimation:

– (void)**setUsesThreadedAnimation:**(BOOL)*flag*

Sets whether the receiver implements animation of the progress indicator in a separate thread. If the application becomes multithreaded as a result of an invocation of this method, the application's performance could become noticeably slower.

See also: – **usesThreadedAnimation**

startAnimation:

– (void)**startAnimation:**(id)*sender*

For an indeterminate progress indicator, starts the animation, which causes the barber pole to start spinning. For a determinate progress indicator, does nothing.

See also: – **animationDelay,** – **stopAnimation:**

stopAnimation:

– (void)**stopAnimation:**(id)*sender*

For an indeterminate progress indicator, stops the animation, which causes the barber pole to stop spinning. For a determinate progress indicator, does nothing.

See also: – **animationDelay,** – **startAnimation:**

usesThreadedAnimation

– (BOOL)**usesThreadedAnimation**

Returns whether the receiver implements the animation of the progress indicator in a separate thread.

See also: – **setUsesThreadedAnimation:**

NSResponder

Inherits From: NSObject

Conforms To: NSCoding
NSObject (NSObject)

Declared In: AppKit/NSResponder.h

Class Description

NSResponder is an abstract class that forms the basis of event and command processing in the Application Kit. The core classes—NSApplication, NSWindow, and NSView—inherit from NSResponder, as must any class that handles events. The responder model is built around three components: event messages, action messages, and the responder chain. An *event message* is a message corresponding directly to an input event, and includes as its sole argument an NSEvent object describing the event; a mouse down or keypress, for example. An *action message* is a higher-level message indicating a command to be performed, which includes as an argument the object requesting the action. Some examples of action messages are the standard **cut:**, **copy:**, and **paste:**.

The *responder chain* is a series of responder objects to which an event or action message is applied. When a given responder object doesn't handle a particular message, the message is passed to its successor in the chain. This allows responder objects to delegate responsibility to other, typically higher-level objects. The responder chain is constructed automatically as described below, but you can insert custom objects into it using the **setNextResponder:** method and examine it with **nextResponder**.

An application can contain any number of responder chains, but only one is active at any given time. It begins with the *first responder* in some NSWindow and proceeds to the NSWindow itself. The first responder is typically the "selected" NSView within the NSWindow, and its next responder is its containing NSView (also called its superview), and so on up to the NSWindow itself. You can safely inject other responders between NSViews, but you can't add responders past the NSWindow. Nearly all event messages apply to a single window's responder chain.

For action messages, a more elaborate responder chain is used, constructed from the individual responder chains of two NSWindows and the application object itself. The NSWindows are the *key window*, whose responder chain gets first crack at action messages, and the *main window*, which follows. The main window is sometimes identical to the key window; the two are typically distinguished when an auxiliary window or panel related to a primary window—such as a Find Panel—is opened. In this case the primary window, which was the key window, becomes the main window, and the Find Panel becomes key. The two windows and the NSApplication object also give their delegates a chance to handle action messages as though they were responders, even though a delegate isn't formally in the responder chain (a **nextResponder** message

to a window or application object doesn't return the delegate). Given all these components, then, the full responder chain comprises these objects:

- The key window's first responder and successors, including objects added with **setNextResponder:**
- The key window itself
- The key window's delegate (which need not inherit from NSResponder)
- The main window's first responder and successors, including objects added with **setNextResponder:**
- The main window itself
- The main window's delegate (which need not inherit from NSResponder)
- The application object, NSApp
- The application object's delegate (which need not inherit from NSResponder)

Selecting the First Responder

The first responder is typically chosen by the user, with the mouse or keyboard. The mechanism by which one object loses its first responder status and another gains it is public though, and you can programmatically change the first responder if necessary. The method that changes the first responder is NSWindow's **makeFirstResponder:**. An NSWindow's first responder is initially itself, though you can set which object will be first responder when the NSWindow is first placed on-screen using the **setInitialFirstResponder:** method.

makeFirstResponder: always asks the current first responder if its ready to resign its status, using **resignFirstResponder**. If the current first responder returns NO when sent this message, **makeFirstResponder:** fails and likewise returns NO. If the current first responder returns YES then the new one is sent a **becomeFirstResponder** message to inform it that it can be the first responder. This object can return NO to reject the assignment, in which case the NSWindow itself becomes the first responder.

When an NSWindow that's the key window receives a mouse-down event, it automatically tries to make first responder the NSView under the event. It does so by asking the NSView whether it wants to become first responder, using the **acceptsFirstResponder** method defined by this class, with the mouse-down event as the argument. This method normally returns NO; responder subclasses that need to be first responder must override it to return YES. This method is also used when the user changes the first responder using the keyboard.

Normally a mouse-down event in a non-key window simply brings the window forward and makes it key, and isn't sent to the NSView over which it occurs. The NSView can claim an initial mouse-down, however, by implementing **acceptsFirstMouse:** to return YES. The argument is the mouse-down event, which the NSView can examine to determine whether it wants to receive the mouse event and potentially become first responder.

An additional consideration for responders that manage selections is of course to set the selection. An NSView that handles mouse events should set this itself. However, objects can also define methods for setting their selection that automatically make the receiver first responder as well. NSTextField's **selectText:**, for example, does something quite like this.

Event and Action Messages in the Responder Chain

The main purpose of the responder chain is to route events and action messages to an appropriate target. Event and action methods are dispatched in different ways, by different methods. Nearly all events enter an application from the Window Server, and are handled automatically by NSApplication's **sendEvent:** method. Action messages are instigated by objects, who use NSApplication's **sendAction:to:from:** method to route them to their proper destinations.

NSApplication's **sendEvent:** analyzes the event and handles some things specially—key equivalents, for example. Most events, however, it passes to the appropriate window for dispatch up its responder chain using NSWindow's **sendEvent:** method. NSResponder's default implementations of all event methods simply pass the message to the next responder, so if no object in the responder chain does anything with the event it's simply lost. As mentioned before, an NSView's next responder is nearly always its superview, so if, for example, the NSView that receives a **mouseDown:** message doesn't handle it, its superview gets a chance, and so on up to the NSWindow. If no object is found to handle the event, the last responder in the chain invokes **noResponderFor:**, which for a key-down event simply beeps. Event-handling objects (subclasses of NSWindow and NSView) can override this method to perform additional steps as needed.

Event messages form a well-known set, so NSResponder provides implementations for all of them. Action messages, however, are defined by custom classes and can't be predicted. For this reason they're dispatched in different manner from events. To instigate an action message, an object invokes NSApplication's **sendAction:to:from:**. The first argument is the selector for the action method to invoke. The second is the intended recipient of the message, often called the *target*. The final argument is usually the object invoking **sendAction:to:from:**, thus indicating which object instigated the action message. If the intended target isn't **nil**, the action is simply sent directly to that object; this is called a *targeted action message*. In the case of an *untargeted action message*, where the target is **nil**, **sendAction:to:from:** searches the full responder chain for an object that implements the action method specified. If it finds one, it sends the message to that object with the instigator of the action message as the sole argument. The receiver of the action message can then use the argument directly as input or query it for additional information. You can find the recipient of an untargeted action message without actually sending the message using **targetForAction:**.

A more general mechanism, which applies to the shorter form of the responder chain, is provided by NSResponder's **tryToPerform:with:**. This method checks the receiver to see if it responds to the selector provided, if so invoking the message. If not, it sends **tryToPerform:with:** to its next responder. NSWindow and NSApplication override this method to include their delegates, but they don't link individual responder chains in the way that NSApplication's **sendAction:to:from:** does. Similar to **tryToPerform:with:** is **doCommandBySelector:**, which takes a method selector and tries to find a responder that implements it. If none is found, the method beeps.

Warning: NSResponder declares a number of action messages, but doesn't actually implement them. You should never send an action message directly to a responder object of an unknown class. Always use NSApplication's **sendAction:to:from:**, NSResponder's **tryToPerform:with:** or **doCommandBySelector:**, or check that the target responds using the NSObject method **respondsToSelector:**.

Implementing Event and Action Methods

Implementing event methods is fairly straightforward. If your subclass handles a particular event, it overrides the method—**keyDown:**, for example—usurping the implementation of its superclass. If your subclass needs to handle particular events some of the time—only some typed characters, perhaps—then it must override the event method to handle the cases it's interested in and to invoke **super**'s implementation otherwise. This allows a superclass to catch the cases it's interested in, and ultimately allows the event to continue on its way along the responder chain if it isn't handled. "Key Events" below describes how to handle keyboard events in your application. See the NSView class specification for information on handling mouse events.

Action methods don't have default implementations, so responder subclasses shouldn't blindly forward action messages to **super**. Passing of action messages is predicated merely on whether an object responds to the method, unlike with the passing of event messages. Of course, if you know that a superclass does in fact implement the method, you can pass it on up from your subclass.

Key Events

Processing keyboard input is by far the most complex part of event handling. The Application Kit goes to great lengths to ease this process for you, and in fact handling the key events that get to your custom objects is fairly straightforward. However, a lot happens to those events on their way from the hardware to the responder chain. The sections below attempt to explain how events are handled through the operating system and the Application Kit, so that you can understand what your objects receive and don't receive.

The Path of a Key Event

Physical keyboard events must pass through the operating system before becoming NSEvent objects in the Application Kit. Depending on the operating system, some of these "raw" events might be trapped before they ever become NSEvent objects. Reserved key combinations are often handled in this way. Key events that arrive at the Application Kit are processed by NSApplication's **sendEvent:** method as indicated before. The application object filters out key equivalents (also known as "Command key events") and sends them out as described under "Key Equivalents and Mnemonics" below. All other key events are passed to the key window's **sendEvent:** method.

The key window first checks the event to see if the Control key is pressed. If it is, the window treats the event as a forced control event, which is blocked from the responder chain and is processed immediately as a potential mnemonic or keyboard interface control event. If this doesn't apply, the event is passed to the window's first responder in a **keyDown:** message, which is how your custom responders receive uninterpreted key events. "Keyboard Input" describes how you can handle these events.

If no view object in the key window accepts the key event, NSWindow's **keyDown:** attempts to handle the key event itself. It tries to interpret the key event as each of the following, in order, beeping if it fails to match any of them to let the user know that the typing couldn't be processed:

- A mnemonic matching the character(s) typed, not requiring the Alternate key to be pressed

- A key equivalent, not requiring the Command (or Control) key to be pressed
- A keyboard interface control event

Key Equivalents and Mnemonics

A key equivalent is a character bound to some view in a window, which causes that view to perform a specified action when the user types that character, usually while pressing the Command key (the Control key on Microsoft Windows). A mnemonic works similarly, using the Alternate key as its cue to action. If both modifier keys are pressed, the key event is interpreted only as a mnemonic. A key equivalent or mnemonic must be a character that can be typed with no modifier keys, or with Shift only. Each is sent down the view hierarchy of a window instead of up the responder chain, but at different times.

Key equivalents are dispatched by the NSApplication object's **sendEvent:** method. On the Mach operating system, this results in a **performKeyEquivalent:** message being sent to every NSWindow in the application until one of them returns YES. On the Microsoft Windows operating system, it results in a **performKeyEquivalent:** message being sent to the menu of the key window, and of the main window if the key window's menu doesn't handle it. This difference in handling means that, among other things, NSWindow subclasses shouldn't override **performKeyEquivalent:**. Also, objects other than menu items shouldn't be assigned key equivalents; they should instead be assigned mnemonics. Key equivalents sent to a window on Mach are passed down the view hierarchy through NSView's abstract implementation of **performKeyEquivalent:**, which forwards the message to each of its subviews until one responds YES, returning NO if none does.

Mnemonics, on the other hand, are dispatched by the key window. If the user presses the Control key as well as the mnemonic's key combination, NSWindow's **sendEvent:** immediately treats that event as a mnemonic to be performed, without sending the event up the responder chain. If the user doesn't press the Control key, the event passes through the window's responder chain, possibly being handled by a responder, before arriving as a **keyDown:** message to the window. In either case, a mnemonic for a top-level menu on Microsoft Windows is sent back to the operating system, and eventually results in the Application Kit invoking a menu item's action. Any other mnemonic is handled by sending a **performMnemonic:** message down the window's view hierarchy, in the same manner as for a **performKeyEquivalent:** message.

Note: **performKeyEquivalent:** takes an NSEvent as its argument, while **performMnemonic:** takes an NSString containing the uninterpreted characters of the key event. You should extract the characters for a key equivalent using NSEvent's **charactersIgnoringModifiers** method.

Keyboard Interface Control

Mnemonics are actually part of a more general means of controlling the user interface via the keyboard. An NSWindow treats certain key events specially, as commands to move control to a different interface object, to simulate a mouse click on it, and so on. In brief, pressing Tab moves control to the next object, whether a button, a text field, or some other kind of control object. Shift-Tab moves control to the previous one. Pressing Space simulates a mouse click for many kinds of control objects, causing a push button to click, a radio button to toggle its state, and so on. In selection lists, pressing Space selects or deselects the

highlighted item; the user can also press Alternate or Shift to extend the selection, not affecting other selected items. Some interface controls also accept arrow-key input.

Each window can be assigned a default button, which is triggered by the Return or Enter key. Also, in modal windows or panels the user can press the Escape key to dismiss the window or panel. If interface control moves to another button, the default button temporarily loses this ability as the user's focus shifts to the button where control resides. However, if control then moves to a different kind of interface object, the default button resumes its normal ability.

The interface objects that are connected together within a window make up the window's *key view loop*. You normally set up the key view loop using Interface Builder, establishing connections to each interface object's **nextKeyView** outlet. You can also set the object that's initially selected when a window is first opened by setting the window's **initialFirstResponder** outlet in Interface Builder. If you do not set this outlet, the window will set a key loop (not necessarily the same as the one you may have specified!) and pick a default initial outlet for you. NSView and NSWindow also define a number of methods for manipulating the key view loop programmatically; see their class specifications for more information.

Keyboard Input

A normal key event eventually makes its way to the responder chain as a **keyDown:** message, which the receiver can handle in any way it sees fit. A text object typically interprets the message as a request to insert text, while a drawing object might only be interested in a few keys, such as Delete and the arrow keys to delete and move selected items. The receiver of a **keyDown:** message can extract the event's characters directly using NSEvent's **characters** or **charactersIgnoringModifiers** methods, or it can pass the key event to the Application Kit's input manager for interpretation according to the user's key bindings. Input management allows key events to be interpreted as text not directly available on the keyboard, such as Kanji and some accented characters, and as commands that affect the content of the responder object handling the event. See the NSInputManager and NSTextInput class and protocol specifications for more information on input management and key binding.

To invoke the input manager, simply invoke NSResponder's **interpretKeyEvents:** message in your implementation of **keyDown:**. This method sends an NSArray of events to the input manager, which interprets the events as text or commands and responds by sending **insertText:** or **doCommandBySelector:** to your responder object. The section "Standard Action Methods for Selecting and Editing" below describes the messages that might be sent to your object.

Standard Action Methods for Selecting and Editing

NSResponder declares prototypes for a number of standard action methods, nearly all related to manipulating selections and editing text. These methods are typically invoked through **doCommandBySelector:** as a result of interpretation by the input manager. They fall into the following general groups:

- Selection movement and expansion
- Text insertion

- General deletion of elements
- Modifying selected text
- Scrolling a document

In most cases the intent of the action method is clear from its name. The individual method descriptions in this specification also provide detailed information about what such a method should normally do. However, a few general concepts apply to many of these methods, and are explained here.

Selection Direction. Some methods refer to spatial directions; left, right, up, down. These are meant to be taken literally, especially in text. To accommodate writing systems with directionality different from Latin script, the terms *forward*, *beginning*, *backward*, and *end* are used.

Selection and insertion point. Methods that refer to moving, deleting, or inserting imply that some elements in the responder are selected, or that there's a zero-length selection at some location (the insertion point). These two things must always be treated consistently. For example, the **insertText:** method is defined as replacing the selection with the text provided. **moveForwardAndModifySelection:** extends or contracts a selection, even if the selection is merely an insertion point. When a selection is modified for the first time, it must always be extended. So a **moveForward...** message extends the selection from its end, while a **moveBackward...** message extends it from its beginning.

Marks. A number of action methods for editing text imitate the Emacs concepts of *point* (the insertion point), and *mark* (an anchor for larger operations normally handled by selections in graphical interfaces). **setMark:** establishes the mark at the current selection, which then remains in effect until the mark is changed again. **selectToMark:** extends the selection to include the mark and all characters between the selection and the mark.

The kill buffer. Also like Emacs, deletion methods affecting lines, paragraphs, and the mark implicitly place the deleted text into a buffer, separate from the pasteboard, from which you can later retrieve it. Methods such as **deleteToBeginningOfLine:** add text to this buffer, and **yank:** replaces the selection with the item in the kill buffer.

Other Uses

The responder chain is utilized by two other mechanisms in the Application Kit. In enabling and disabling a menu item, the application object consults the full responder chain for an object that implements the menu item's action method, as described in the NSMenuActionResponder protocol specification. Similarly, the Services facility passes **validRequestorForSendType:returnType:** messages along the full responder chain to check for objects that are eligible for services offered by other applications. The Services validation process is described fully in "Services" in *OPENSTEP Programming Topics*.

Adopted Protocols

NSCoding

– encodeWithCoder:
– initWithCoder:

Method Types

Changing the first responder

– acceptsFirstResponder
– becomeFirstResponder
– resignFirstResponder

Setting the next responder

– setNextResponder:
– nextResponder

Event methods

– mouseDown:
– mouseDragged:
– mouseUp:
– mouseMoved:
– mouseEntered:
– mouseExited:
– rightMouseDown:
– rightMouseDragged:
– rightMouseUp:
– keyDown:
– keyUp:
– flagsChanged:
– helpRequested:

Special key event methods

– interpretKeyEvents:
– performKeyEquivalent:
– performMnemonic:

Clearing key events

– flushBufferedKeyEvents

Action methods

– capitalizeWord:
– centerSelectionInVisibleArea:
– changeCaseOfLetter:
– complete:
– deleteBackward:
– deleteForward:
– deleteToBeginningOfLine:
– deleteToBeginningOfParagraph:
– deleteToEndOfLine:
– deleteToEndOfParagraph:
– deleteToMark:
– deleteWordBackward:
– deleteWordForward:
– indent:
– insertBacktab:
– insertNewline:
– insertNewlineIgnoringFieldEditor:
– insertParagraphSeparator:
– insertTab:
– insertTabIgnoringFieldEditor:
– insertText:
– lowercaseWord:
– moveBackward:
– moveBackwardAndModifySelection:
– moveDown:
– moveDownAndModifySelection:
– moveForward:
– moveForwardAndModifySelection:
– moveLeft:
– moveRight:
– moveToBeginningOfDocument:
– moveToBeginningOfLine:
– moveToBeginningOfParagraph:
– moveToEndOfDocument:
– moveToEndOfLine:
– moveToEndOfParagraph:
– moveUp:
– moveUpAndModifySelection:
– moveWordBackward:
– moveWordBackwardAndModifySelection:
– moveWordForward:
– moveWordForwardAndModifySelection:

– pageDown:
– pageUp:
– scrollLineDown:
– scrollLineUp:
– scrollPageDown:
– scrollPageUp:
– selectAll:
– selectLine:
– selectParagraph:
– selectToMark:
– selectWord:
– setMark:
– showContextHelp:
– swapWithMark:
– transpose:
– transposeWords:
– uppercaseWord:
– yank:

Dispatch methods

– doCommandBySelector:
– tryToPerform:with:

Terminating the responder chain

– noResponderFor:

Services menu updating

– validRequestorForSendType:returnType:

Setting the menu

– setMenu:
– menu

Setting the interface style

– setInterfaceStyle:
– interfaceStyle

Instance Methods

acceptsFirstResponder

– (BOOL)**acceptsFirstResponder**

Overridden by subclasses to return YES if the receiver can handle key events and action messages sent up the responder chain. NSResponder's implementation returns NO, indicating that by default a responder

object doesn't agree to become first responder. Objects that aren't first responder can receive mouse event messages, but no other event or action messages.

See also: – **becomeFirstResponder**, – **resignFirstResponder**, – **needsPanelToBecomeKey** (NSView)

becomeFirstResponder

– (BOOL)**becomeFirstResponder**

Notifies the receiver that it's about to become first responder in its NSWindow. NSResponder's implementation returns YES, accepting first responder status. Subclasses can override this method to update state or perform some action such as highlighting the selection, or to return NO, refusing first responder status.

Use NSWindow's **makeFirstResponder:**, not this method, to make an object the first responder. Never invoke this method directly.

See also: – **resignFirstResponder**, – **acceptsFirstResponder**

capitalizeWord:

– (void)**capitalizeWord:**(id)*sender*

Implemented by subclasses to capitalize the word or words surrounding the insertion point or selection, expanding the selection if necessary. If either end of the selection partially covers a word, that entire word is made lowercase. NSResponder declares, but doesn't implement this method.

See also: – **lowercaseWord:**, – **uppercaseWord:**, – **changeCaseOfLetter:**

centerSelectionInVisibleArea:

– (void)**centerSelectionInVisibleArea:**(id)*sender*

Implemented by subclasses to scroll the selection, whatever it is, inside its visible area. NSResponder declares, but doesn't implement this method.

See also: – **scrollLineDown:**, – **scrollLineUp:**, – **scrollPageDown:**, – **scrollPageUp:**

changeCaseOfLetter:

– (void)**changeCaseOfLetter:**(id)*sender*

Implemented by subclasses to change the case of a letter or letters in the selection, perhaps by opening a panel with capitalization options or by cycling through possible case combinations. NSResponder declares, but doesn't implement this method.

See also: – **lowercaseWord:**, – **uppercaseWord:**, – **capitalizeWord:**

complete:

– (void)**complete:**(id)*sender*

Implemented by subclasses to complete an operation in progress or a partially constructed element. This can be interpreted, for example, as a request to attempt expansion of a partial word, such as for expanding a glossary shortcut, or to close a graphic item being drawn. NSResponder declares, but doesn't implement this method.

deleteBackward:

– (void)**deleteBackward:**(id)*sender*

Implemented by subclasses to delete the selection if there is one, or a single element backward from the insertion point (a letter or character in text, for example). NSResponder declares, but doesn't implement this method.

deleteForward:

– (void)**deleteForward:**(id)*sender*

Implemented by subclasses to delete the selection if there is one, or a single element forward from the insertion point (a letter or character in text, for example). NSResponder declares, but doesn't implement this method.

deleteToBeginningOfLine:

– (void)**deleteToBeginningOfLine:**(id)*sender*

Implemented by subclasses to delete the selection if there is one, or all text from the insertion point to the beginning of a line (typically of text). Also places the deleted text into the kill buffer. NSResponder declares, but doesn't implement this method.

See also: – **yank:**

deleteToBeginningOfParagraph:

– (void)**deleteToBeginningOfParagraph:**(id)*sender*

Implemented by subclasses to delete the selection if there is one, or all text from the insertion point to the beginning of a paragraph of text. Also places the deleted text into the kill buffer. NSResponder declares, but doesn't implement this method.

See also: – **yank:**

deleteToEndOfLine:

– (void)**deleteToEndOfLine:**(id)*sender*

Implemented by subclasses to delete the selection if there is one, or all text from the insertion point to the end of a line (typically of text). Also places the deleted text into the kill buffer. NSResponder declares, but doesn't implement this method.

deleteToEndOfParagraph:

– (void)**deleteToEndOfParagraph:**(id)*sender*

Implemented by subclasses to delete the selection if there is one, or all text from the insertion point to the end of a paragraph of text. Also places the deleted text into the kill buffer. NSResponder declares, but doesn't implement this method.

See also: – **yank:**

deleteToMark:

– (void)**deleteToMark:**(id)*sender*

Implemented by subclasses to delete the selection if there is one, or all items from the insertion point to a previously placed mark, including the selection itself if not empty. Also places the deleted text into the kill buffer. NSResponder declares, but doesn't implement this method.

See also: – **setMark:**, – **selectToMark:**, – **yank:**

deleteWordBackward:

– (void)**deleteWordBackward:**(id)*sender*

Implemented by subclasses to delete the selection if there is one, or a single word backward from the insertion point. NSResponder declares, but doesn't implement this method.

deleteWordForward:

– (void)**deleteWordForward:**(id)*sender*

Implemented by subclasses to delete the selection if there is one, or a single word forward from the insertion point. NSResponder declares, but doesn't implement this method.

doCommandBySelector:

– (void)**doCommandBySelector:**(SEL)*aSelector*

Attempts to perform the method indicated by *aSelector*. The method should take a single argument of type **id** and return **void**. If the receiver responds to *aSelector*, it invokes the method with **nil** as the argument. If the receiver doesn't respond, it sends this message to its next responder with the same selector. NSWindow and NSApplication also send the message to their delegates. If the receiver has no next responder or delegate, it beeps.

See also: – **tryToPerform:with:**, – **sendAction:to:from:** (NSApplication)

flagsChanged:

– (void)**flagsChanged:**(NSEvent *)*theEvent*

Informs the receiver that the user has pressed or released a modifier key (Shift, Control, and so on). NSResponder's implementation simply passes this message to the next responder.

flushBufferedKeyEvents

– (void)**flushBufferedKeyEvents**

Overridden by subclasses to clear any unprocessed key events.

helpRequested:

– (void)**helpRequested:**(NSEvent *)*theEvent*

Displays context-sensitive help for the receiver if such exists, otherwise passes this message to the next responder. NSWindow invokes this method automatically when the user clicks for help. Subclasses need not override this method, and application code shouldn't directly invoke it.

See also: – **showContextHelp:**

indent:

– (void)**indent:**(id)*sender*

Implemented by subclasses to indent the selection or the insertion point if there is no selection. NSResponder declares, but doesn't implement this method.

insertBacktab:

– (void)**insertBacktab:**(id)*sender*

Implemented by subclasses to handle a "backward tab." A field editor might respond to this by selecting the field before it, while a regular text object either doesn't respond to, or ignores such a message. NSResponder declares, but doesn't implement this method.

insertNewline:

– (void)**insertNewline:**(id)*sender*

Implemented by subclasses to insert a line-break character at the insertion point or selection, deleting the selection if there is one, or to end editing if the receiver is a text field or other field editor. NSResponder declares, but doesn't implement this method.

insertNewlineIgnoringFieldEditor:

– (void)**insertNewlineIgnoringFieldEditor:**(id)*sender*

Implemented by subclasses to insert a line-break character at the insertion point or selection, deleting the selection if there is one. Unlike **insertNewline:**, this method always inserts a line-break character and doesn't cause the receiver to end editing. NSResponder declares, but doesn't implement this method.

insertParagraphSeparator:

– (void)**insertParagraphSeparator:**(id)*sender*

Implemented by subclasses to insert a paragraph separator at the insertion point or selection, deleting the selection if there is one. NSResponder declares, but doesn't implement this method.

insertTab:

– (void)**insertTab:**(id)*sender*

Implemented by subclasses to insert a tab character at the insertion point or selection, deleting the selection if there is one, or to end editing if the receiver is a text field or other field editor. NSResponder declares, but doesn't implement this method.

insertTabIgnoringFieldEditor:

– (void)**insertTabIgnoringFieldEditor:**(id)*sender*

Implemented by subclasses to insert a tab character at the insertion point or selection, deleting the selection if there is one. Unlike **insertTab:**, this method always inserts a tab character and doesn't cause the receiver to end editing. NSResponder declares, but doesn't implement this method.

insertText:

– (void)**insertText:**(NSString *)*aString*

Overridden by subclasses to insert *aString* at the insertion point or selection, deleting the selection if there is one. NSResponder's implementation simply passes this message to the next responder, or beeps if there is no next responder.

interfaceStyle

– (NSInterfaceStyle)**interfaceStyle**

Returns the receiver's interface style. **interfaceStyle** is an abstract method in NSResponder and just returns NSNoInterfaceStyle. It is overridden in classes such as NSWindow and NSView to return the interface style, such as NSMacintoshInterfaceStyle or NSWindows95InterfaceStyle. A responder's style (if other than NSNoInterfaceStyle) overrides all other settings, such as those established by the defaults system.

See also: – **setInterfaceStyle:**

interpretKeyEvents:

– (void)**interpretKeyEvents:**(NSArray *)*eventArray*

Invoked by subclasses from their **keyDown:** method to handle a series of key events. This method sends the character input in *eventArray* to the system input manager for interpretation as text to insert or commands to perform. The input manager responds to the request by sending **insertText:** and **doCommandBySelector:** messages back to the invoker of this method. Subclasses shouldn't override this method.

See the NSInputManager and NSTextInput class and protocol specifications for more information on input management.

keyDown:

– (void)**keyDown:**(NSEvent *)*theEvent*

Informs the receiver that the user has pressed a key. The receiver can interpret *theEvent* itself, or pass it to the system input manager using **interpretKeyEvents:**. NSResponder's implementation simply passes this message to the next responder.

keyUp:

– (void)**keyUp:**(NSEvent *)*theEvent*

Informs the receiver that the user has released a key. NSResponder's implementation simply passes this message to the next responder.

lowercaseWord:

– (void)**lowercaseWord:**(id)*sender*

Implemented by subclasses to make lowercase every letter in the word or words surrounding the insertion point or selection, expanding the selection if necessary. If either end of the selection partially covers a word, that entire word is made lowercase. NSResponder declares, but doesn't implement this method.

See also: – **uppercaseWord:**, – **capitalizeWord:**, – **changeCaseOfLetter:**

menu

– (NSMenu *)**menu**

Returns the receiver's menu. For NSApplication this is the same as the menu returned by its **mainMenu** method.

See also: – **setMenu:**, – **menuForEvent:** (NSView), + **defaultMenu** (NSView)

mouseDown:

– (void)**mouseDown:**(NSEvent *)*theEvent*

Informs the receiver that the user has pressed the left mouse button. NSResponder's implementation simply passes this message to the next responder.

mouseDragged:

– (void)**mouseDragged:**(NSEvent *)*theEvent*

Informs the receiver that the user has moved the mouse with the left button pressed. NSResponder's implementation simply passes this message to the next responder.

mouseEntered:

– (void)**mouseEntered:**(NSEvent *)*theEvent*

Informs the receiver that the mouse has entered a tracking rectangle. NSResponder's implementation simply passes this message to the next responder.

mouseExited:

– (void)**mouseExited:**(NSEvent *)*theEvent*

Informs the receiver that the mouse has exited a tracking rectangle. NSResponder's implementation simply passes this message to the next responder.

mouseMoved:

– (void)**mouseMoved:**(NSEvent *)*theEvent*

Informs the receiver that the mouse has moved. NSResponder's implementation simply passes this message to the next responder.

See also: – **setAcceptsMouseMovedEvents:** (NSWindow)

mouseUp:

– (void)**mouseUp:**(NSEvent *)*theEvent*

Informs the receiver that the user has released the left mouse button. NSResponder's implementation simply passes this message to the next responder.

moveBackward:

– (void)**moveBackward:**(id)*sender*

Implemented by subclasses to move the selection or insertion point one element or character backward. In text, if there is a selection it should be deselected, and the insertion point should be placed at the beginning of the former selection. NSResponder declares, but doesn't implement this method.

moveBackwardAndModifySelection:

– (void)**moveBackwardAndModifySelection:**(id)*sender*

Implemented by subclasses to expand or reduce either end of the selection backward by one element or character. If the end being modified is the backward end, this method expands the selection; if the end being modified is the forward end, it reduces the selection. The first **moveBackwardAndModifySelection:** or **moveForwardAndModifySelection:** method in a series determines the end being modified by always expanding. Hence, this method results in the backward end becoming the mobile one if invoked first.

NSResponder declares, but doesn't implement this method.

moveDown:

– (void)**moveDown:**(id)*sender*

Implemented by subclasses to move the selection or insertion point one element or character down. In text, if there is a selection it should be deselected, and the insertion point should be placed below the beginning of the former selection. NSResponder declares, but doesn't implement this method.

moveDownAndModifySelection:

– (void)**moveDownAndModifySelection:**(id)*sender*

Implemented by subclasses to expand or reduce the top or bottom end of the selection downward by one element, character, or line (whichever is appropriate for text direction). If the end being modified is the bottom, this method expands the selection; if the end being modified is the top, it reduces the selection. The first **moveDownAndModifySelection:** or **moveUpAndModifySelection:** method in a series determines the end being modified by always expanding. Hence, this method results in the bottom end becoming the mobile one if invoked first.

NSResponder declares, but doesn't implement this method.

moveForward:

– (void)**moveForward:**(id)*sender*

Implemented by subclasses to move the selection or insertion point one element or character forward. In text, if there is a selection it should be deselected, and the insertion point should be placed at the end of the former selection. NSResponder declares, but doesn't implement this method.

moveForwardAndModifySelection:

– (void)**moveForwardAndModifySelection:**(id)*sender*

Implemented by subclasses to expand or reduce either end of the selection forward by one element or character. If the end being modified is the backward end, this method reduces the selection; if the end being modified is the forward end, it expands the selection. The first **moveBackwardAndModifySelection:** or **moveForwardAndModifySelection:** method in a series determines the end being modified by always expanding. Hence, this method results in the forward end becoming the mobile one if invoked first.

NSResponder declares, but doesn't implement this method.

moveLeft:

– (void)**moveLeft:**(id)*sender*

Implemented by subclasses to move the selection or insertion point one element or character to the left. In text, if there is a selection it should be deselected, and the insertion point should be placed at the left end of the former selection. NSResponder declares, but doesn't implement this method.

moveRight:

– (void)**moveRight:**(id)*sender*

Implemented by subclasses to move the selection or insertion point one element or character to the right. In text, if there is a selection it should be deselected, and the insertion point should be placed at the right end of the former selection. NSResponder declares, but doesn't implement this method.

moveToBeginningOfDocument:

– (void)**moveToBeginningOfDocument:**(id)*sender*

Implemented by subclasses to move the selection to the first element of the document, or the insertion point to the beginning. NSResponder declares, but doesn't implement this method.

moveToBeginningOfLine:

– (void)**moveToBeginningOfLine:**(id)*sender*

Implemented by subclasses to move the selection to the first element of the selected line, or the insertion point to the beginning of the line. NSResponder declares, but doesn't implement this method.

moveToBeginningOfParagraph:

– (void)**moveToBeginningOfParagraph:**(id)*sender*

Implemented by subclasses to move the insertion point to the beginning of the selected paragraph. NSResponder declares, but doesn't implement this method.

moveToEndOfDocument:

– (void)**moveToEndOfDocument:**(id)*sender*

Implemented by subclasses to move the selection to the last element of the document, or the insertion point to the end. NSResponder declares, but doesn't implement this method.

moveToEndOfLine:

– (void)**moveToEndOfLine:**(id)*sender*

Implemented by subclasses to move the selection to the last element of the selected line, or the insertion point to the end of the line. NSResponder declares, but doesn't implement this method.

moveToEndOfParagraph:

– (void)**moveToEndOfParagraph:**(id)*sender*

Implemented by subclasses to move the insertion point to the end of the selected paragraph. NSResponder declares, but doesn't implement this method.

moveUp:

– (void)**moveUp:**(id)*sender*

Implemented by subclasses to move the selection or insertion point one element or character up. In text, if there is a selection it should be deselected, and the insertion point should be placed above the beginning of the former selection. NSResponder declares, but doesn't implement this method.

moveUpAndModifySelection:

– (void)**moveUpAndModifySelection:**(id)*sender*

Implemented by subclasses to expand or reduce the top or bottom end of the selection upward by one element, character, or line (whichever is appropriate for text direction). If the end being modified is the bottom, this method reduces the selection; if the end being modified is the top, it expands the selection. The

first **moveDownAndModifySelection:** or **moveUpAndModifySelection:** method in a series determines the end being modified by always expanding. Hence, this method results in the top end becoming the mobile one if invoked first.

NSResponder declares, but doesn't implement this method.

moveWordBackward:

– (void)**moveWordBackward:**(id)*sender*

Implemented by subclasses to move the selection or insertion point one word backward. If there is a selection it should be deselected, and the insertion point should be placed at the end of the first word preceding the former selection. NSResponder declares, but doesn't implement this method.

moveWordBackwardAndModifySelection:

– (void)**moveWordBackwardAndModifySelection:**(id)*sender*

Implemented by subclasses to expand or reduce either end of the selection backward by one whole word. If the end being modified is the backward end, this method expands the selection; if the end being modified is the forward end, it reduces the selection. The first **moveWordBackwardAndModifySelection:** or **moveWordForwardAndModifySelection:** method in a series determines the end being modified by always expanding. Hence, this method results in the backward end becoming the mobile one if invoked first.

NSResponder declares, but doesn't implement this method.

moveWordForward:

– (void)**moveWordForward:**(id)*sender*

Implemented by subclasses to move the selection or insertion point one word forward. If there is a selection it should be deselected, and the insertion point should be placed at the beginning of the first word following the former selection. NSResponder declares, but doesn't implement this method.

moveWordForwardAndModifySelection:

– (void)**moveWordForwardAndModifySelection:**(id)*sender*

Implemented by subclasses to expand or reduce either end of the selection forward by one whole word. If the end being modified is the backward end, this method reduces the selection; if the end being modified is the forward end, it expands the selection. The first **moveWordBackwardAndModifySelection:** or **moveWordForwardAndModifySelection:** method in a series determines the end being modified by always expanding. Hence, this method results in the forward end becoming the mobile one if invoked first.

NSResponder declares, but doesn't implement this method.

nextResponder

– (NSResponder *)**nextResponder**

Returns the receiver's next responder, or **nil** if it has none.

See also: – **setNextResponder:**, – **noResponderFor:**

noResponderFor:

– (void)**noResponderFor:**(SEL)*eventSelector*

Handles the case where an event or action message falls off the end of the responder chain. NSResponder's implementation beeps if *eventSelector* is **keyDown:**.

pageDown:

– (void)**pageDown:**(id)*sender*

Implemented by subclasses to scroll the receiver down (or back) one page in its scroll view, also moving the insertion point to the top of the newly-displayed page. NSResponder declares, but doesn't implement this method.

See also: – **scrollPageDown:**, – **scrollPageUp:**

pageUp:

– (void)**pageUp:**(id)*sender*

Implemented by subclasses to scroll the receiver up (or forward) one page in its scroll view, also moving the insertion point to the top of the newly-displayed page. NSResponder declares, but doesn't implement this method.

See also: – **scrollPageDown:**, – **scrollPageUp:**

performKeyEquivalent:

– (BOOL)**performKeyEquivalent:**(NSEvent *)*theEvent*

Overridden by subclasses to handle a key equivalent. If the character code or codes in *theEvent* match the receiver's key equivalent, the receiver should respond to the event and return YES. NSResponder's implementation does nothing and returns NO.

Note: **performKeyEquivalent:** takes an NSEvent as its argument, while **performMnemonic:** takes an NSString containing the uninterpreted characters of the key event. You should extract the characters for a key equivalent using NSEvent's **charactersIgnoringModifiers** method.

See also: – **performKeyEquivalent:** (NSView), – **performKeyEquivalent:** (NSButton)

performMnemonic:

– (BOOL)**performMnemonic:**(NSString *)*aString*

Overridden by subclasses to handle a mnemonic. If the character code or codes in *theEvent* match the receiver's mnemonic, the receiver should respond to the event and return YES. NSResponder's implementation does nothing and returns NO.

See also: – **performMnemonic:** (NSView)

resignFirstResponder

– (BOOL)**resignFirstResponder**

Notifies the receiver that it's been asked to relinquish its status as first responder in its NSWindow. NSResponder's implementation returns YES, resigning first responder status. Subclasses can override this method to update state or perform some action such as unhighlighting the selection, or to return NO, refusing to relinquish first responder status.

Use NSWindow's **makeFirstResponder:**, not this method, to make an object the first responder. Never invoke this method directly.

See also: – **becomeFirstResponder**, – **acceptsFirstResponder**

rightMouseDown:

– (void)**rightMouseDown:**(NSEvent *)*theEvent*

Informs the receiver that the user has pressed the right mouse button. NSResponder's implementation simply passes this message to the next responder.

rightMouseDragged:

– (void)**rightMouseDragged:**(NSEvent *)*theEvent*

Informs the receiver that the user has moved the mouse with the right button pressed. NSResponder's implementation simply passes this message to the next responder.

rightMouseUp:

– (void)**rightMouseUp:**(NSEvent *)*theEvent*

Informs the receiver that the user has released the right mouse button. NSResponder's implementation simply passes this message to the next responder.

scrollLineDown:

– (void)**scrollLineDown:**(id)*sender*

Implemented by subclasses to scroll the receiver one line down in its scroll view, without changing the selection. NSResponder declares, but doesn't implement this method.

See also: – **scrollPageDown:**, – **lineScroll** (NSScrollView)

scrollLineUp:

– (void)**scrollLineUp:**(id)*sender*

Implemented by subclasses to scroll the receiver one line up in its scroll view, without changing the selection. NSResponder declares, but doesn't implement this method.

See also: – **scrollPageUp:**, – **lineScroll** (NSScrollView)

scrollPageDown:

– (void)**scrollPageDown:**(id)*sender*

Implemented by subclasses to scroll the receiver one page down in its scroll view, without changing the selection. NSResponder declares, but doesn't implement this method.

See also: – **pageDown:**, – **pageUp:**, – **pageScroll** (NSScrollView)

scrollPageUp:

– (void)**scrollPageUp:**(id)*sender*

Implemented by subclasses to scroll the receiver one page up in its scroll view, without changing the selection. NSResponder declares, but doesn't implement this method.

See also: – **pageDown:**, – **pageUp:**, – **pageScroll** (NSScrollView)

selectAll:

– (void)**selectAll:**(id)*sender*

Implemented by subclasses to select all selectable elements. NSResponder declares, but doesn't implement this method.

selectLine:

– (void)**selectLine:**(id)*sender*

Implemented by subclasses to select all elements in the line or lines containing the selection or insertion point. NSResponder declares, but doesn't implement this method.

selectParagraph:

– (void)**selectParagraph:**(id)*sender*

Implemented by subclasses to select all paragraphs containing the selection or insertion point. NSResponder declares, but doesn't implement this method.

selectToMark:

– (void)**selectToMark:**(id)*sender*

Implemented by subclasses to select all items from the insertion point or selection to a previously placed mark, including the selection itself if not empty. NSResponder declares, but doesn't implement this method.

See also: – **setMark:**, – **deleteToMark:**

selectWord:

– (void)**selectWord:**(id)*sender*

Implemented by subclasses to extend the selection to the nearest word boundaries outside it (up to, but not including, word delimiters). NSResponder declares, but doesn't implement this method.

setInterfaceStyle:

– (void)**setInterfaceStyle:(**NSInterfaceStyle**)***interfaceStyle*

Sets the receiver's style to the style specified by *interfaceStyle*, such as NSMacintoshInterfaceStyle or NSWindows95InterfaceStyle. **setInterfaceStyle:** is an abstract method in NSResponder, but is overridden

in classes such as NSWindow and NSView to actually set the interface style. You should almost never need to invoke or override this method, but if you do override it, your version should always invoke **super**.

See also: – **interfaceStyle**

setMark:

– (void)**setMark:**(id)*sender*

Implemented by subclasses to set a mark at the insertion point or selection, which is used by **deleteToMark:** and **selectToMark:**. NSResponder declares, but doesn't implement this method.

See also: – **swapWithMark:**

setMenu:

– (void)**setMenu:**(NSMenu *)*aMenu*

Sets the receiver's menu to *aMenu*. For NSApplication this is the same as the main menu, typically set using **setMainMenu:**.

See also: – **menu**

setNextResponder:

– (void)**setNextResponder:**(NSResponder *)*aResponder*

Sets the receiver's next responder to *aResponder*.

See also: – **nextResponder**

showContextHelp:

– (void)**showContextHelp:**(id)*sender*

Implemented by subclasses to invoke the host platform's help system, displaying information relevant to the receiver and its current state.

See also: – **helpRequested:**

swapWithMark:

– (void)**swapWithMark:**(id)*sender*

Swaps the mark and the selection or insertion point, so that what was marked is now the selection or insertion point, and what was the insertion point or selection is now the mark. NSResponder declares, but doesn't implement this method.

See also: – **setMark:**

transpose:

– (void)**transpose:**(id)*sender*

Transposes the characters to either side of the insertion point and advances the insertion point past both of them. Does nothing to a selected range of text. NSResponder declares, but doesn't implement this method.

transposeWords:

– (void)**transposeWords:**(id)*sender*

NSResponder declares, but doesn't implement this method.

tryToPerform:with:

– (BOOL)**tryToPerform:**(SEL)*anAction* **with:**(id)*anObject*

Attempts to perform the action method indicated by *anAction*. The method should take a single argument of type **id** and return **void**. If the receiver responds to *anAction*, it invokes the method with *anObject* as the argument and returns YES. If the receiver doesn't respond, it sends this message to its next responder with the same selector and object. Returns NO if no responder is found that responds to *anAction*.

See also: – **doCommandBySelector:**, – **sendAction:to:from:** (NSApplication)

uppercaseWord:

– (void)**uppercaseWord:**(id)*sender*

Implemented by subclasses to make uppercase every letter in the word or words surrounding the insertion point or selection, expanding the selection if necessary. If either end of the selection partially covers a word, that entire word is made uppercase. NSResponder declares, but doesn't implement this method.

See also: – **lowercaseWord:**, – **capitalizeWord:**, – **changeCaseOfLetter:**

validRequestorForSendType:returnType:

– (id)**validRequestorForSendType:**(NSString *)*sendType* **returnType:**(NSString *)*returnType*

Overridden by subclasses to determine what services are available. With each event, and for each service in the Services menu, the application object sends this message up the responder chain with the send and return type for the service being checked. This method is therefore invoked many times per event. If the receiver can place data of *sendType* on the pasteboard and receive data of *returnType*, it should return **self**; otherwise it should return **nil**. NSResponder's implementation simply forwards this message to the next responder, ultimately returning **nil**.

Either *sendType* or *returnType*—but not both—may be empty. If *sendType* is empty, the service doesn't require input from the application requesting the service. If *returnType* is empty, the service doesn't return data.

See "Services" in *OPENSTEP Programming Topics* for more information.

See also: – **registerServicesMenuSendTypes:returnTypes:** (NSApplication),
– **writeSelectionToPasteboard:types:** (NSServicesRequests protocol),
– **readSelectionFromPasteboard:** (NSServicesRequests protocol)

yank:

– (void)**yank:**(id)*sender*

Replaces the insertion point or selection with text from the kill buffer. If invoked sequentially, cycles through the kill buffer in reverse order. See "Standard Action Methods for Selecting and Editing" in the class description for more information on the kill buffer. NSResponder declares, but doesn't implement this method.

See also: – **deleteToBeginningOfLine:**, – **deleteToEndOfLine:**, – **deleteToBeginningOfParagraph:**,
– **deleteToEndOfParagraph:**, – **deleteToMark:**

NSRulerMarker

Inherits From: NSObject

Conforms To: NSCopying
NSObject (NSObject)

Declared In: AppKit/NSRulerMarker.h

Class Description

An NSRulerMarker displays a symbol on an NSRulerView, indicating a location for whatever graphic element it represents in the client of the NSRulerView (for example, a margin or tab setting, or the edges of a graphic on the page). A ruler marker comprises three primary attributes: the image it displays on the NSRulerView, the location of that image, and the object it represents. The **setImage:**, **setMarkerLocation:** and **setRepresentedObject:** methods set each of these attributes, respectively. In addition, a ruler marker records an offset for the image, allowing it to be placed relative to the marker location much in the way a cursor's hot spot relates a cursor image to the mouse location; the **setImageOrigin:** method establishes this offset.

Most of these attributes are set upon initialization by the **initWithRulerView:markerLocation:image: imageOrigin:** method. New ruler markers don't have represented objects; the client typically establishes the represented object in its **rulerView:didAddMarker:** method. A new NSRulerMarker can be moved around in its NSRulerView, but not removed from it. The **setMovable:** and **setRemovable:** methods alter these default settings.

Represented objects allow the NSRulerView's client to distinguish among different attributes of the selection. In the NSRulerView client methods, the client can retrieve the marker's represented object to determine what attribute to alter. Generic attributes can be represented by string or other value objects, such as the edge names "Left", "Right", "Top", and "Bottom". Attributes already implemented as objects can be represented by those objects. For example, the text system records tab stops as NSTextTab objects, which include the tab location and its alignment. When an NSTextView is the client view of a ruler, it simply makes the NSTextTabs the represented objects of the ruler markers.

Adopted Protocols

NSCopying

– copyWithZone:

Method Types

Creating instances

– initWithRulerView:markerLocation:image:imageOrigin:

Getting the ruler view

– ruler

Setting the image

– setImage:
– image
– setImageOrigin:
– imageOrigin
– imageRectInRuler
– thicknessRequiredInRuler

Setting movability

– setMovable:
– isMovable
– setRemovable:
– isRemovable

Setting the location

– setMarkerLocation:
– markerLocation

Setting the represented object

– setRepresentedObject:
– representedObject

Drawing and event handling

– drawRect:
– isDragging
– trackMouse:adding:

Instance Methods

drawRect:

– (void)**drawRect:**(NSRect)*aRect*

Draws the part of the receiver's image that intersects *aRect* in the NSRulerView's coordinate system.

See also: – **imageRectInRuler**

image

– (NSImage *)**image**

Returns the NSImage object displayed by the receiver.

See also: – **setImage:**

imageOrigin

– (NSPoint)**imageOrigin**

Returns the point in the receiver's image that's positioned at the receiver's location on the NSRulerView, expressed in the image's coordinate system.

For a horizontal ruler, the *x* coordinate of the image origin is aligned with the location of the marker, and the *y* coordinate lies on the baseline of the ruler. For vertical rulers, the *y* coordinate of the image origin is the location, and the *x* coordinate lies on the baseline.

See also: – **setImageOrigin:**, – **imageRectInRuler**

imageRectInRuler

– (NSRect)**imageRectInRuler**

Returns the rectangle occupied by the marker's image, in the NSRulerView's coordinate system, accounting for whether the NSRulerView's coordinate system is flipped.

See also: – **drawRect:**, – **thicknessRequiredInRuler**

initWithRulerView:markerLocation:image:imageOrigin:

– (id)**initWithRulerView:**(NSRulerView *)*aRulerView*
markerLocation:(float)*location*
image:(NSImage *)*anImage*
imageOrigin:(NSPoint)*imageOrigin*

Initializes a newly allocated NSRulerMarker object, associating it with (but not adding it to) *aRulerView* and assigning the attributes provided. *location* is the *x* or *y* position of the marker in the client view's coordinate system, depending on whether the NSRulerView is horizontal or vertical. *anImage* is the image displayed at the marker location, and *imageOrigin* is the point within the image that's positioned at the marker location, expressed in pixels relative to the lower-left corner of the image. This method raises an NSInvalidArgumentException if *aRulerView* or *anImage* is **nil**.

Note: The image used to draw the marker must be appropriate for the orientation of the ruler. Markers may need to look different on a horizontal ruler than on a vertical ruler, and the NSRulerView neither scales nor rotates the images.

To add the new ruler marker to *aRulerView*, use either of NSRulerView's **addMarker:** or **trackMarker: withMouseEvent:** methods. **addMarker:** immediately puts the marker on the ruler, while **trackMarker: withMousEvent:** allows the client view to intercede in the addition and placement of the marker.

A new ruler marker is can be moved on its NSRulerView, but not removed. Use **setMovable:** and **setRemovable:** to change these attributes. The new ruler marker also has no represented object; use **setRepresentedObject:** to provide or change it.

This method retains *anImage*, since it's an essential part of the ruler marker, but doesn't retain *aRulerView* (the NSRulerView instead retains the new marker when it's added). This method is the designated initializer for the NSRulerMarker class. Returns **self**.

See also: – **setMarkerLocation:**, – **setImage:**, – **setImageOrigin:**

isDragging

– (BOOL)**isDragging**

Returns YES if the receiver is being dragged, NO otherwise.

See also: – **trackMouse:adding:**

isMovable

– (BOOL)**isMovable**

Returns YES if the user can move the receiver on its NSRulerView, NO otherwise. NSRulerMarkers are by default movable.

See also: – **setMovable:**, – **isRemovable**

isRemovable

– (BOOL)**isRemovable**

Returns YES if the user can remove the receiver from its NSRulerView, NO otherwise. NSRulerMarkers cannot by default be removed from their NSRulerViews.

See also: – **setRemovable:**, – **isMovable**

markerLocation

– (float)**makerLocation**

Returns the location of the receiver in the coordinate system of the NSRulerView's client view. This is an *x* position for a horizontal ruler, a *y* position for a vertical ruler.

See also: – **setMarkerLocation:**

representedObject

– (id <NSCopying>)**representedObject**

Returns the object that the receiver represents, as explained in the class description.

See also: – **setRepresentedObject:**

ruler

– (NSRulerView *)**ruler**

Returns the NSRulerView that the receiver belongs to.

See also: – **addMarker:** (NSRulerView)

setImage:

– (void)**setImage:**(NSImage *)*anImage*

Sets the receiver's image to *anImage*.

See also: – **image**, – **setImageOrigin:**

setImageOrigin:

– (void)**setImageOrigin:**(NSPoint)*aPoint*

Sets the point in the receiver's image that's positioned at the receiver's location on the NSRulerView to that at *aPoint*. This point is always expressed in pixels relative to the lower-left corner of the image.

For a horizontal ruler, the *x* coordinate of the image origin is aligned with the location of the marker, and the *y* coordinate lies on the baseline of the ruler. For vertical rulers, the *y* coordinate of the image origin is the location, and the *x* coordinate lies on the baseline.

See also: – **imageOrigin**, – **setImage:**, – **setMarkerLocation:**

setMarkerLocation:

– (void)**setMarkerLocation:**(float)*location*

Sets the location of the receiver in the coordinate system of the NSRulerView's client view to *location*. This is an *x* position for a horizontal ruler, a *y* position for a vertical ruler.

See also: – **markerLocation**, – **setImageOrigin:**

setMovable:

– (void)**setMovable:**(BOOL)*flag*

Controls whether the user can move the receiver in its NSRulerView. If *flag* is YES, the user can drag the marker image in the ruler. If *flag* is NO, the receiver is immovable. NSRulerMarkers are by default movable.

See also: – **isMovable**, – **setRemovable:**

setRemovable:

– (void)**setRemovable:**(BOOL)*flag*

Controls whether the user can remove the receiver from its NSRulerView. If *flag* is YES, the user can drag the marker image off of the ruler. If *flag* is NO, the receiver can't be removed. NSRulerMarkers are by default not removable.

See also: – **isRemovable**, – **setMovable:**

setRepresentedObject:

– (void)**setRepresentedObject:**(id <NSCopying>)*anObject*

Sets the object that the receiver represents to *anObject*. See the class description for more information on the represented object.

See also: – **representedObject**

thicknessRequiredInRuler

– (float)**thicknessRequiredInRuler**

Returns the amount of the receiver's image that's displayed above or to the left of the NSRulerView's baseline, the height for a horizontal ruler or width for a vertical ruler.

See also: – **imageOrigin**

trackMouse:adding:

– (BOOL)**trackMouse:**(NSEvent *)*theEvent* **adding:**(BOOL)*flag*

Handles user manipulation of the receiver in its NSRulerView. NSRulerView invokes this method automatically to add a new marker or to move or remove an existing marker. You should never need to invoke it directly.

If *flag* is YES, the receiver is a new marker being added to its NSRulerView. Before the receiver actually adds itself to the NSRulerView, it queries the NSRulerView's client view using **rulerView: shouldAddMarker:**. If the client view responds to this method and returns NO, this method immediately returns NO and the new marker isn't added.

If *flag* is NO, this method attempts to move or remove an existing marker, once again based on responses from the NSRulerView's client view. If the receiver is neither movable nor removable, this method immediately returns NO. Further, if the NSRulerView's client responds to **rulerView: shouldMoveMarker:** and returns NO, this method returns NO, indicating that the receiver can't be moved.

If the receiver is being added or moved, this method queries the client view using **rulerView: willAddMarker:atLocation:** or **rulerView:willMoveMarker:toLocation:**, respectively. If the client responds to the method, the return value is used as the receiver's location. These methods are invoked repeatedly as the receiver is dragged within the NSRulerView.

If the receiver is an existing marker being removed (dragged off the ruler), this method queries the client view using **rulerView:shouldRemoveMarker:**. If the client responds to this method and returns NO, the marker is pinned to the NSRulerView's baseline (following the mouse on the baseline if it's movable).

When the user releases the mouse, this method informs the client view of the marker's new status using **rulerView:didAddMarker:**, **rulerView:didMoveMarker:**, or **rulerView:didRemoveMarker:** as appropriate. The client view can use this notification to set the marker's represented object, modify its state and redisplay (for example, adjusting text layout around a new tab stop), or take whatever other action it might need. If *flag* is YES and the user dragged the new marker away from the ruler, the marker isn't added, no message is sent, and this method returns NO.

See the NSRulerView class description for more information on these client methods.

See also: – **isMovable**, – **isRemovable**

NSRulerView

Inherits From: NSView : NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSRulerView.h

Class at a Glance

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

Coordinate System	Used for
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

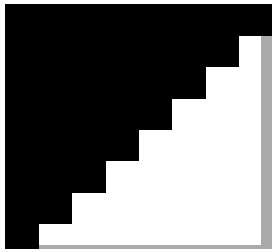
If the client has a custom accessory view, it sets that using **setAccessoryView:**. Clients without accessory views should avoid removing the ruler view's accessory view when appropriating the ruler, as this can cause unsightly screen flicker as the ruler is redrawn. It's better in this case for a client 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.

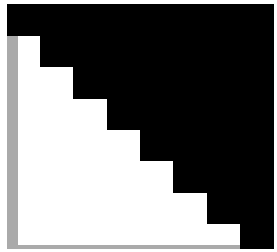
Setting Ruler Markers

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 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 in "Handling Mouse Events in the Ruler Area" below. 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 the "Preparing a Ruler View for Use" heading. 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:abbreviation: unitToPointsConversionFactor:stepUpCycle:stepDownCycle:**

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: – **ruleThickness**, – **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: – **thicknessRequiredInRuler** (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:**, – **setHorizontalRulerView:** (NSScrollView), – **setVerticalRulerView:** (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:**, – **thicknessRequiredInRuler** (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**, – **setHorizontalRulerView:** (NSScrollView), – **setVerticalRulerView:** (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.

NSSavePanel

Inherits From:	Windows: NSObject Mach: NSPanel : NSWindow : NSResponder : NSObject
Conforms To:	Windows: NSObject (NSObject) Mach: NSCoding (NSResponder) NSObject (NSObject)
Declared In:	AppKit/NSSavePanel.h

Class at a Glance

Purpose

An NSSavePanel object manages a panel that allows users to specify the directory and name under which a file is saved. It supports browsing of the file system and it accommodates custom accessory views. An NSSavePanel

is a recycled object: when you request a Save panel, NSSavePanel tries to reuse an existing Save panel rather than create a new one.

Principal Attributes

Delegate	Browser
Form	Prompt
Title	File name
Directory	Accessory view

Creation

+ savePanel (class method)

Commonly Used Methods

runModal	Displays the panel and begins the event loop.
filename	Returns the selected or entered file name.
directory	Returns the full path of the selected file.
ok:	Invoked when users click OK.

Class Description

NSSavePanel creates and manages a Save panel, and allows you to run the panel in a modal loop. The Save panel provides a simple way for a user to specify a file to use when saving a document or other data. It can restrict the user to files of a certain type, as specified by a file name extension.

When the user decides on a file name, the message **panel:isValidFilename:** is sent to the NSSavePanel's delegate. If it responds to that message, the delegate can determine whether the specified file name can be used; it returns YES if the file name is valid, or NO if the Save panel should stay up and wait for the user to type in a different file name.

Typically, you access an NSSavePanel by invoking the **savePanel** method. When the class receives a **savePanel** message, it tries to reuse an existing panel rather than create a new one. When a panel is reused its attributes are reset to the default values so that the effect is the same as receiving a new panel. Because

a Save panel may be reused, you shouldn't modify the instance returned by **savePanel** except through the methods listed below. For example, you can set the panel's title and required file type, but not the arrangement of the buttons within the panel. If you must modify the Save panel substantially, create and manage your own instance using the **alloc...** and **init...** methods rather than the **savePanel** method.

A typical programmatic use of NSSavePanel requires you to:

- Invoke **savePanel.**
- Configure the panel (for instance, set its title or add a custom view).
- Run the panel in a model loop.
- Test the result; if successful, save the file under the chosen name and in the chosen directory.

The following code fragment demonstrates this sequence. (Two objects in this example, **newView** and **textData**, are assumed to be defined and created elsewhere.)

```
NSSavePanel *sp;
int runResult;

/* create or get the shared instance of NSSavePanel */
sp = [NSSavePanel savePanel];

/* set up new attributes */
[sp setAccessoryView:newView];
[sp setRequiredFileType:@"txt"];

/* display the NSSavePanel */
runResult = [sp runModalForDirectory:NSHomeDirectory() file:@""];

/* if successful, save file under designated name */
if (runResult == NSOKButton) {
    if (![textData writeToFile:[sp filename] atomically:YES])
         NSBeep();
}
```

Method Types

Creating an NSSavePanel

+ savePanel

Customizing the NSSavePanel

– setAccessoryView:
– accessoryView
– setTitle:
– title
– setPrompt:
– prompt

Setting directory and file type

– setDirectory:
– setRequiredFileType:
– requiredFileType
– treatsFilePackagesAsDirectories
– setTreatsFilePackagesAsDirectories:
– validateVisibleColumns

Running the NSSavePanel

– runModal
– runModalForDirectory:file:

Getting user selections

– directory
– filename

Action methods

– cancel:
– ok:

Responding to user input

– selectText:

Setting the delegate

– setDelegate:

Class Methods

savePanel

+ (NSSavePanel *)**savePanel**

Returns an instance of NSSavePanel, creating one if necessary. Otherwise, the instance is a recycled NSSavePanel object. The method sets the attributes of the instance to the default values:

- Current working directory as starting point
- Prompt of “Name”
- No required file types
- File packages not treated as directories
- No delegate
- No accessory view

Instance Methods

accessoryView

– (NSView *****)**accessoryView**

Returns the custom accessory view for the current application. This view is set by **setAccessoryView:**.

See also: – **setAccessoryView:**

cancel:

– (void)**cancel:***(id)sender*

Invoked when the user clicks the panel's Cancel button.

directory

– (NSString *)**directory**

Returns the absolute pathname of the directory currently shown in the panel. Do not invoke this method within a modal session (**runModal** or **runModalForDirectory:file:**) because the directory information is only updated just before the modal session ends.

See also: – **setDirectory:**

encodeWithCoder:

– (void)**encodeWithCoder:**(NSCoder *)*coder*

Overrides the superclass implementation of this NSCoding protocol method to raise an exception. The NSSavePanel does not get encoded.

See also: – **initWithCoder:**

filename

– (NSString *)**filename**

Returns the absolute path name of the file currently shown in the panel. Do not invoke this method within a modal session (**runModal** or **runModalForDirectory:file:**) because the filename information is only updated just before the modal session ends.

initWithCoder:

– (id)**initWithCoder:**(NSCoder *)*coder*

Overrides the superclass implementation of this NSCoding protocol method to raise an exception. The NSSavePanel does not get decoded.

See also: – **encodeWithCoder:**

ok:

– (void)**ok:***(id)sender*

Invoked when the user clicks the panel's OK button.

prompt

– (NSString *)**prompt**

Returns the prompt of the Save panel field that holds the current pathname or file name. By default this prompt is "Name:".

See also: – **setPrompt:**

requiredFileType

– (NSString *)**requiredFileType**

Returns the required file type (if any). A file specified in the Save panel is saved with the designated file name and this file type as an extension. Examples of common file types are "rtf", "tiff", and "ps". An empty NSString return value indicates that the user can save to any ASCII file.

See also: – **setRequiredFileType:**

runModal

– (int)**runModal**

Displays the panel and begins its event loop with the current working (or last selected) directory as the default starting point. Invokes **runModalForDirectory:file:** (file argument is an empty NSString), which in turn performs NSApplication's **runModalForWindow:** method with **self** as the argument. Returns NSOKButton (if the user clicks the OK button) or NSCancelButton (if the user clicks the Cancel button).

Do not invoke **filename** or **directory** within a modal loop because the information that these methods fetch is updated only upon return.

See also: – **runModalForDirectory:file:**, – **runModalForWindow:** (NSApplication)

runModalForDirectory:file:

– (int)**runModalForDirectory:**(NSString *)*path* **file:**(NSString *)*filename*

Initializes the panel to the directory specified by *path* and, optionally, the file specified by *filename*, then displays it and begins its modal event loop; *path* and *filename* can be empty strings, but cannot be **nil**. The method invokes Application's **runModalForWindow:** method with **self** as the argument. Returns NSOKButton (if the user clicks the OK button) or NSCancelButton (if the user clicks the Cancel button). Do not invoke **filename** or **directory** within a modal loop because the information that these methods fetch is updated only upon return.

See also: – **runModal**, – **runModalForWindow:** (Application)

selectText:

– (void)**selectText:***(id)sender*

Advances the current browser selection one line when Tab or the up-arrow key is pressed, and goes back one line when Shift-Tab or the down-arrow key is pressed; after it makes the new selection it writes the selected item in the field after the prompt. The argument *sender* identifies the object invoking this method. This method is primarily of interest to those who want to override it to get different behavior.

setAccessoryView:

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

Customizes the panel for the application by adding a custom NSView (aView) to the panel. The custom NSView that's added appears just above the OK and Cancel buttons at the bottom of the panel. The NSSavePanel automatically resizes itself to accommodate *aView*. You can invoke this method repeatedly to change the accessory view as needed. If *aView* is **nil**, the NSSavePanel removes the current accessory view.

See also: – **accessoryView**

setDelegate:

– (void)**setDelegate:**(id)*anObject*

Makes *anObject* the NSSavePanel's delegate, after verifying which delegate methods are implemented. Use NSWindow's **delegate** method to retrieve the NSSavePanel's delegate.

setDirectory:

– (void)**setDirectory:**(NSString *)*path*

Sets the current path name in the Save panel's browser. The *path* argument must be an absolute path name.

See also: – **directory**

setPrompt:

– (void)**setPrompt:**(NSString *)*prompt*

Sets the prompt of the field that holds the current pathname or file name. This prompt appears on all NSSavePanels (or all NSOpenPanels if the receiver of this message is an NSOpenPanel) in your application. "Name:" is the default prompt string.

See also: – **prompt**

setRequiredFileType:

– (void)**setRequiredFileType:**(NSString *)*type*

Specifies the *type*, a file name extension to be appended to any selected files that don't already have that extension; "nib" and "rtf" are examples. The argument *type* should not include the period that begins the extension. You need to invoke this method each time the Save panel is used for another file type within the application.

See also: – **requiredFileType**

setTreatsFilePackagesAsDirectories:

– (void)**setTreatsFilePackagesAsDirectories:**(BOOL)*flag*

Sets the NSSavePanel's behavior for displaying file packages (for example, MyApp.app) to the user. If *flag* is YES, the user is shown files and subdirectories within a file package. If NO, the NSSavePanel shows each file package as a file, thereby giving no indication that it is a directory.

See also: – **treatsFilePackagesAsDirectories**

setTitle:

– (void)**setTitle:**(NSString *)*title*

Sets the title of the NSSavePanel to *title*. By default, “Save” is the title string. If you adapt the NSSavePanel for other uses, its title should reflect the user action that brings it to the screen.

See also: – **title**

title

- (NSString *)**title**

Returns the title of the NSSavePanel.

See also: **– setTitle:**

treatsFilePackagesAsDirectories

– (BOOL)**treatsFilePackagesAsDirectories**

Use to determine whether the Save panel displays file packages to the user as directories. Returns YES if the user is shown files and subdirectories within a file package; returns NO (the default) if the user is shown only file-package names, with no indication that they are directories.

See also: – **setTreatsFilePackagesAsDirectories:**

validateVisibleColumns

– (void)**validateVisibleColumns**

Validates and possibly reloads the browser columns visible in the Save panel by causing the delegate method **panel:shouldShowFilename:** to be invoked. One situation in which this method would find use is whey you want the browser show only files with certain extensions based on the selection made in an accessory-view pop-up list. When the user changes the selection, you would invoke this method to revalidate the visible columns.

Methods Implemented by the Delegate

panel:compareFilename:with:caseSensitive:

– (NSComparisonResult)**panel:***(id)sender*
compareFilename:(NSString *)*fileName1*
with:(NSString *)*fileName2*
caseSensitive:(BOOL)*flag*

Controls the ordering of files presented by the NSSavePanel. This method should return:

- NSOrderedAscending if *fileName1* should precede *fileName2*
- NSOrderedSame if the two names are equivalent
- NSOrderedDescending if *fileName2* should precede *fileName1*

The *flag* argument, if YES, indicates that the ordering is to be case-sensitive.

Don't reorder file names in the Save panel without good reason, since it may confuse the user to have files in one Save panel or Open panel ordered differently than those in other such panels or in the Workspace Manager. The default behavior of Save and Open panel is to order files as they appear in the Workspace Manager file viewer. Note also that by implementing this method you will reduce the operating performance of the panel.

panel:shouldShowFilename:

– (BOOL)**panel:(id)***sender*
shouldShowFilename:*(NSString *)filename*

The NSSavePanel sends this message to the panel's delegate for each file or directory (*filename*) it is about to load in the browser. This method gives the delegate the opportunity to filter out items that it doesn't want the user to see or choose. The delegate returns YES if *filename* should be displayed, and NO if the NSSavePanel should ignore the file or directory.

panel:isValidFilename:

– (BOOL)**panel:(id)*sender***
isValidFilename:*(NSString *)filename*

The NSSavePanel sends this message just before the end of a modal session for each file name displayed or selected (including file names in multiple selections). The delegate determines whether it wants the file identified by *filename*; it returns YES if the file name is valid, or NO if the NSSavePanel should stay in its modal loop and wait for the user to type in or select a different file name or names. If the delegate refuses a file name in a multiple selection, none of the file names in the selection are accepted.

NSScreen

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: AppKit/NSScreen.h

Class Description

An NSScreen object describes the attributes of a computer's monitor, or screen. An application may use an NSScreen object to retrieve information about a screen and use this information to decide what to display upon that screen. For example, an application may use the **deepestScreen** method to find out which of the available screens can best represent color and then may choose to display all of its windows on that screen.

The two main attributes of a screen are its depth and its dimensions. The **depth** method describes the screen depth (such as two-bit, eight-bit, or twelve-bit) and tells you if the screen can display color. The **frame** method gives the screen's dimensions and location as an NSRect.

The device description dictionary contains more complete information about the screen. Use NSScreen's **deviceDescription** method to access the dictionary, and use these keys to retrieve information about a screen:

Dictionary Key	Value
NSDeviceResolution	An NSValue that contains an NSSize which indicates the screen's resolution in dots per inch (dpi).
NSDeviceColorSpaceName	The screen's color space name. See the NSGraphics class specification for a list of possible values.
NSDeviceBitsPerSample	An NSNumber containing an integer that indicates the bit depth of screen images (2-bit, 8-bit, and so on).
NSDeviceIsScreen	"YES" (a string), indicating the device is a screen.
NSDeviceSize	An NSValue that contains an NSSize which indicates the screen's size in points.

The device description dictionary contains information about not only screens, but all other system devices such as printers and windows. There are other keys into the dictionary that you would use to obtain information about these other devices. For a complete list of device dictionary keys, see NSGraphics.h.

The application object should be created before you use the methods in this class, so that the application object can make the necessary connection to the Window System. You can make sure the application object exists by invoking NSApplication's **sharedApplication** method, which creates it if necessary. If you created your application with Project Builder, the application object is automatically created for you in **main()**.

Method Types

Getting NSScreens	
	+ mainScreen
	+ deepestScreen
	+ screens
Reading screen information	
	– depth
	– frame
	– supportedWindowDepths
	– deviceDescription
	– visibleFrame

Class Methods

deepestScreen

+ (NSScreen *)**deepestScreen**

Returns an NSScreen object representing the screen that can best represent color. This method always returns an object, even if there is only one screen and it is not a color screen.

mainScreen

+ (NSScreen *)**mainScreen**

Returns an NSScreen object representing the main screen. The main screen is the screen with the key window.

screens

+ (NSArray *)**screens**

Returns an array of NSScreen objects representing all of the screens available on the system. Raises NSWindowServerCommunicationException if the screens information can’t be obtained from the window system.

Instance Methods

depth

– (NSWindowDepth)**depth**

Returns the screen's depth, including whether the screen can display color.

deviceDescription

– (NSDictionary *)**deviceDescription**

Returns the device dictionary as described in the class description.

frame

– (NSRect)**frame**

Returns the dimensions and location of the screen in an NSRect.

supportedWindowDepths

– (const NSWindowDepth *)**supportedWindowDepths**

Returns a zero-terminated array of the window depths supported by the screen.

visibleFrame

– (NSRect)**visibleFrame**

Returns the dimensions and location of the visible screen in an NSRect. The frame for the visible screen is adjusted according to the interface style. For example, on the Macintosh, the visible screen area does not include the menu bar.

NSScroller

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

Class at a Glance

Purpose

An NSScroller object is a user control for scrolling a document view within a container view. You normally don't need to program with NSScrollers, as Interface Builder allows you to fully configure them with an NSScrollView.

Principal Attributes

- Scrolling by small and large increments
- Proportional knob showing visible amount of document

Creation

Interface Builder

– initWithFrame:	Initializes the NSScroller.

Commonly Used Methods

– hitPart	Indicates where the user clicked the NSScroller.
– floatValue (NSControl)	Returns the position of the NSScroller's knob.
– setFloatValue:knobProportion:	Sets the position and size of the NSScroller's knob.

Class Description

An NSScroller controls scrolling of a document view within an NSScrollView's clip view (or potentially another kind of container view). It typically displays a pair of buttons that the user can click to scroll by a small amount (called a line increment or decrement) and Alternate-click to scroll by a large amount (called a page increment or decrement), plus a slot containing a knob that the user can drag directly to the desired location. The knob indicates both the position within the document view and, by varying in size within the slot, the amount visible relative to the size of the document view. You can configure whether an NSScroller uses scroll buttons, but it always draws the knob when there's room for it.

Interface Builder automatically associates NSScrollers with an NSScrollView, allowing you to configure most aspects of scrolling behavior without programming. If you create one programmatically using **initWithFrame:** the new NSScroller automatically collapses the shorter of its two dimensions to the standard scroller width, as returned by the **scrollerWidth** class method. In this manner a tall, narrow frame results in a vertical NSScroller and a short, wide frame results in a horizontal NSScroller.

Don't use an NSScroller when an NSSlider would be better. A slider represents a range of values for something in the application and lets the user choose a setting. A scroller represents the relative position of the visible portion of a view and lets the user choose which portion to view.

Interaction with a Container View

NSScroller is a public class primarily for programmers who decide not to use an NSScrollView but who want to present a consistent user interface. Its use outside of NSScrollViews is discouraged except in cases where the porting of an existing application is made more straightforward. Setting up an NSScroller with a custom container view class (or a completely different kind of target) involves establishing the standard target and action as defined by NSControl, and implementing the target's action method appropriately.

As an NSScroller tracks the mouse, it sends an action message to its target with itself as the argument. The target then performs the scrolling operation based on these things:

- The orientation of the NSScroller, vertical or horizontal, which determines the axis to scroll along. This can be done by getting the NSScroller's frame and determining the longer dimension, or by keeping a reference to designated vertical and horizontal scrollers (as NSScrollView does).
- The direction and scale of scrolling. NSScroller's **hitPart** method returns this information, described in more detail below.
- If the hit part is the knob or its slot, the NSScroller's value, which indicates where to position the document view in the container view. The **floatValue** method provides this information, which the target should interpret relative to the size of the document view's frame minus the size of the container view's bounds.

As indicated above, the direction and scale of a scrolling operation is determined by the value returned from a **hitPart** message, which indicates the various parts of an NSScroller in terms of their intended result (not in terms of their location or appearance):

Value	How to Scroll
NSScrollerKnob	Directly to the NSScroller's value, as given by **floatValue**
NSScrollerKnobSlot	Directly to the NSScroller's value, as given by **floatValue**
NSScrollerDecrementLine	Up or left by a small amount
NSScrollerDecrementPage	Up or left by a large amount
NSScrollerIncrementLine	Down or right by a small amount
NSScrollerIncrementPage	Down or right by a large amount
NSScrollerNoPart	Don't scroll at all

Note: These part codes are interpreted differently depending on the method you use them with. See the individual method descriptions for **hitPart**, **rectForPart:**, and **testPart:** for details.

The four decrement/increment values require the target to calculate an appropriate amount to scroll by. Line and page amounts are up to the target or the container view to define. NSScrollView, for example, allows you to set these in Interface Builder or with its **setLineScroll:** and **setPageScroll:** methods. Once the target has scrolled the document view by a decrement or increment, it should update the NSScroller's position using **setFloatValue:**.

The container view or target should also keep tabs on its size and on the size and position of its document view. Any time these change it should update its NSScrollers using **setFloatValue:knobProportion:**. NSClipView, for example, overrides most of NSView's **setBounds...** and **setFrame...** methods to perform this updating.

Method Types

Determining NSScroller size

+ scrollerWidth

Laying out an NSScroller

– setArrowsPosition:
– arrowsPosition

Setting the knob position

– setFloatValue:knobProportion:
– knobProportion

Calculating layout

– rectForPart:
– testPart:
– checkSpaceForParts
– usableParts

Drawing the parts

– drawArrow:highlight:
– drawKnob
– drawParts
– highlight:

Event handling

– hitPart
– trackKnob:
– trackScrollButtons:

Class Methods

scrollerWidth

+ (float)**scrollerWidth**

Returns the width of instances. NSScrollView uses this value to lay out its components. Subclasses that use a different width should override this method.

Instance Methods

arrowsPosition

– (NSScrollArrowPosition)**arrowsPosition**

Returns the location of the scroll buttons within the receiver, as described under **setArrowsPosition:**.

checkSpaceForParts

– (void)**checkSpaceForParts**

Checks to see if there is enough room in the receiver to display the knob and buttons. **usableParts** returns the state calculated by this method. You should never need to invoke this method; it's invoked automatically whenever the NSScroller's size changes.

drawArrow:highlight:

– (void)**drawArrow:**(NSScrollerArrow)*arrow* **highlight:**(BOOL)*flag*

Draws the scroll button indicated by *arrow*, which is either NSScrollerIncrementArrow (the down or right scroll button) or NSScrollerDecrementArrow (up or left). If *flag* is YES, the button is drawn highlighted, otherwise it's drawn normally. You should never need to invoke this method directly, but may wish to override it to customize the appearance of scroll buttons.

See also: – **drawKnob**, – **rectForPart:**

drawKnob

– (void)**drawKnob**

Draws the knob. You should never need to invoke this method directly, but may wish to override it to customize the appearance of the knob.

See also: – **drawArrow:highlight:**, – **rectForPart:**

drawParts

– (void)**drawParts**

Caches images for the scroll buttons and knob. It's invoked only once when the NSScroller is created. You may want to override this method if you alter the look of the NSScroller, but you should never invoke it directly.

highlight:

– (void)**highlight:**(BOOL)*flag*

Highlights or unhighlights the scroll button that the user clicked. The receiver invokes this method while tracking the mouse; you should not invoke it directly. If *flag* is YES, the appropriate part is drawn highlighted, otherwise it's drawn normally.

See also: – **drawArrow:highlight:**, – **rectForPart:**

hitPart

– (NSScrollerPart)**hitPart**

Returns a part code indicating the manner in which the scrolling should be performed:

Value	How to Scroll
NSScrollerKnob	Directly to the NSScroller's value, as given by **floatValue**
NSScrollerKnobSlot	Directly to the NSScroller's value, as given by **floatValue**
NSScrollerDecrementLine	Up or left by a small amount
NSScrollerDecrementPage	Up or left by a large amount
NSScrollerIncrementLine	Down or right by a small amount
NSScrollerIncrementPage	Down or right by a large amount
NSScrollerNoPart	Don't scroll at all

This method is typically invoked by an NSScrollView to determine how to scroll its document view when it receives an action message from the NSScroller.

initWithFrame

– (id)**initWithFrame:**(NSRect)*frameRect*

Initializes the receiver as normal, but collapsing *frameRect*'s narrower dimension to the value returned by the **scrollerWidth** method. This enforces the appearance of vertical and horizontal NSScrollers. This is the designated initializer for the NSScroller class. Returns **self**.

knobProportion

– (float)**knobProportion**

Returns the portion of the knob slot that the knob should fill, as a floating-point value from 0.0 (minimal size) to 1.0 (fills the slot).

rectForPart:

– (NSRect)**rectForPart:**(NSScrollerPart)*aPart*

Returns the rectangle occupied by *aPart*, which for this method is interpreted literally rather than as an indicator of scrolling direction:

Value	Part
NSScrollerKnob	The knob itself
NSScrollerKnobSlot	The slot that the knob moves in
NSScrollerDecrementLine	The up or left scroll button
NSScrollerDecrementPage	The region of the slot above or to the left of the knob
NSScrollerIncrementLine	The down or right scroll button
NSScrollerIncrementPage	The region of the slot below or to the right of the knob

Note the interpretations of NSScrollerDecrementPage and NSScrollerIncrementPage. The actual part of an NSScroller that causes page-by-page scrolling varies from platform to platform, so as a convenience these part codes refer to useful parts different from the scroll buttons.

Returns NSZeroRect if the part requested isn't present on the receiver.

See also: – **hitPart**, – **testPart:**, – **usableParts**

setArrowsPosition:

– (void)**setArrowsPosition:**(NSScrollArrowPosition)*location*

Sets the location of the scroll buttons within the Scroller to *location*, or inhibits their display, as follows:

Value	Meaning
NSScrollerArrowsMaxEnd	Buttons at bottom or right
NSScrollerArrowsMinEnd	Buttons at top or left
NSScrollerArrowsNone	No buttons

Note: On Microsoft Windows scroll buttons appear at either end of the scroller rather than both on one end. A value other than NSScrollerArrowsNone is thus reinterpreted on that platform to display the buttons at either end.

See also: – **arrowsPosition**

setFloatValue:knobProportion:

– (void)**setFloatValue:**(float)*aFloat* **knobProportion:**(float)*knobProp*

Sets the position of the knob to *aFloat*, which is a value between 0.0 (indicating the top or left end) and 1.0 (the bottom or right end). Also sets the proportion of the knob slot filled by the knob to *knobProp*, also a value between 0.0 (minimal size) and 1.0 (fills the slot).

See also: – **floatValue** (NSControl), – **knobProportion**

testPart:

– (NSScrollerPart)**testPart:**(NSPoint)*aPoint*

Returns the part that would be hit by a mouse-down event at *aPoint* (expressed in the receiver's coordinate system):

Return Value	Part Identified
NSScrollerKnob	The knob itself
NSScrollerKnobSlot	The slot that the knob moves in (returned only if there's no knob)
NSScrollerDecrementLine	The up or left scroll button
NSScrollerDecrementPage	The region of the slot above or to the left of the knob
NSScrollerIncrementLine	The down or right scroll button
NSScrollerIncrementPage	The region of the slot below or to the right of the knob
NSScrollerNoPart	(*aPoint* isn't in the NSScroller)

Note the interpretations of NSScrollerDecrementPage and NSScrollerIncrementPage. The actual part of an NSScroller that causes page-by-page scrolling varies from platform to platform, so as a convenience these part codes refer to useful parts different from the scroll buttons.

See also: – **hitPart**, – **rectForPart:**

trackKnob:

– (void)**trackKnob:**(NSEvent *)*theEvent*

Tracks the knob and sends action messages to the receiver's target. This method is invoked automatically when the NSScroller receives a mouse-down event in the knob; you should not invoke it directly.

trackScrollButtons:

– (void)**trackScrollButtons:**(NSEvent *)*theEvent*

Tracks the scroll buttons and sends action messages to the receiver's target. This method is invoked automatically when the NSScroller receives a mouse-down event in a scroll button; you should not invoke this method directly.

usableParts

– (NSUsableScrollerParts)**usableParts**

Returns a value indicating which parts of the receiver are displayed and usable. This is one of:

Value	Meaning
NSNoScrollerParts	Scroller has neither a knob nor scroll buttons, only the knob slot.
NSOnlyScrollerArrows	Scroller has only scroll buttons, no knob.
NSAllScrollerParts	Scroller has at least a knob, possibly also scroll buttons.

See also: – **checkSpaceForParts**, – **arrowsPosition**

NSScrollView

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

Class at a Glance

Purpose

An NSScrollView allows the user to scroll a document view that's too large to display in its entirety. In addition to the document view, it displays horizontal and vertical scrollers and rulers (depending on which it's configured to have).

Principal Attributes

- Configurable scrollers
- Configurable rulers
- Displays a special cursor over its document view
- Small and large increment scrolling
- Dynamic (continuous) scrolling

Creation

Interface Builder

– initWithFrame:	Designated initializer.

Commonly Used Methods

– setDocumentView:	Sets the cursor used over the document view.
– setLineScroll:	Sets the amount by which the document view moves during scrolling.
– setRulersVisible:	Displays or hides rulers.

Class Description

The NSScrollView class is the central coordinator for the Application Kit's scrolling machinery, composed of this class, NSClipView, and NSScroller. An NSScrollView displays a portion of a *document view* that's too large to be displayed whole, and provides NSScrollers that allow the user to move the document view within the NSScrollView. An NSScrollView can be configured with a vertical scroller, a horizontal scroller,

or both. In addition to the basic accoutrements, an NSScrollView keeps a cursor that it sets whenever the mouse is over its document view, and maintains both horizontal and vertical ruler objects that can be hidden and displayed.

An NSScrollView encloses its document view within an NSClipView, using this view to actually position and monitor the document view. Because the NSClipView manages the content of the NSScrollView, it's also called the *content view*. The content view positions the document view by altering its bounds rectangle, which determines where the document view's frame lies. The content view also monitors changes in the document view's size and notifies the NSScrollView so that the scrollers can be updated to reflect the new size. The **documentView** and **contentView** methods return an NSScrollView's major component views.

NSScrollView defines three levels of scrolling: by line, by page, and direct. Scrolling by line moves the document view by a small amount, typically when the user clicks the scroll buttons of a scroller. Scrolling by page moves the document view by a larger amount, typically near the size of the content view, when the user Alternate-clicks the scroll buttons, and on some platforms in the slot of the scroller. You set these amounts using **setLineScroll:** and **setPageScroll:**, respectively (Interface Builder also lets you set these directly). Direct scrolling moves the document view to the position of the scroller's knob as the user drags it. This either displays the document view continuously as it scrolls or displays it only when the user releases the mouse, as configured with the **setScrollsDynamically:** method.

When created programmatically, an NSScrollView has no scrollers. You can set them up using the **setHasVerticalScroller:** and **setHasHorizontalScroller:** methods with an argument of YES, which cause the NSScrollView to allocate and maintain instances of the NSScroller class. You can substitute specialized scrollers using the **setVerticalScroller:** and **setHorizontalScroller:** methods. Note that in any case you must use the **setHas...** methods to make sure the NSScrollView displays its scrollers.

NSView contains a few additional methods for scrolling a document programmatically. To scroll a specific point to the upper left corner of the view, use **scrollPoint:**. To scroll a specific area into view, use **scrollRectToVisible:**. To scroll automatically as the user drags something outside the window, use **autoscroll:**.

Rulers

An NSScrollView can be set to hold both horizontal and vertical rulers using the **setHasHorizontalRuler:** and **setHasVerticalRuler:** methods. These allocate instances of the NSRulerView class, but unlike with scrollers don't immediately display the rulers. To do this, use the **setRulersVisible:** method. You can substitute custom ruler objects using **setHorizontalRulerView:** and **setVerticalRulerView:**, and to customize the rulers for all instances of NSScrollView you can set the class used with **setRulerViewClass:** . This causes all subsequent rulers created by NSScrollViews to be of the class you specify.

An NSScrollView's rulers don't automatically establish the document view as their client. The document view itself (or a subview) is responsible, as its selection and other state changes, for retrieving the rulers using NSScrollView's **horizontalRulerView** and **verticalRulerView** methods and for establishing itself as the client using NSRulerView's **setClientView:** method.

How Scrolling Works

As indicated above, an NSScrollView's document view is actually positioned by the content view, which sets its bounds rectangle in such a way that the document view's frame moves relative to it. However, the action sequence between the scrollers and the NSScrollView and the manner in which scrolling is performed involve a bit more detail than this.

Scrolling typically occurs because of user actions on an NSScroller object, which sends the NSScrollView a private action message telling it to scroll based on the NSScroller's state. This process is described in the class description for the NSScroller class under "Interaction with a Container View." If you plan to implement your own kind of scrolling view or scroller object, you should read that section.

NSClipView's **scrollToPoint:** is the method that actually scrolls the document view. It essentially translates the origin of the content view's bounds rectangle, but it also optimizes redisplay by copying as much of the rendered document view as remains visible, and only asking the document view to draw newly exposed regions. This usually improves scrolling performance, but may not always be appropriate behavior. You can turn it off using NSClipView's **setCopiesOnScroll:** method. If you do leave copy-on-scroll active, be sure to scroll the document view programmatically using **scrollToPoint:** rather than **translateOriginToPoint:**.

Whether the document view scrolls explicitly through a user action or an NSClipView message, or implicitly through a **setFrame:** or other such message, the content view monitors it closely. Whenever the document view's frame or bounds rectangle changes, it informs the NSScrollView of the change with a **reflectScrolledClipView:** message. This method updates the NSScroller objects to reflect the position and size of the visible portion of the document view. You may find on occasion that you need to invoke this method explicitly when manipulating the document view directly.

Autoscrolling

In addition to user-driven and programmatic scrolling, you can program any NSView to automatically scroll when the user drags the mouse outside the enclosing NSClipView. This allows the user to drag an item in order to move it, and have the document view automatically shift itself in the appropriate direction when the user drags the item past the visible area. NSClipView's **autoscroll:** method takes an NSEvent object of the mouse-dragged type and scrolls its document view in the opposite direction from the mouse location, making the portion of the document view that would be under the mouse become visible. NSView also implements **autoscroll:** to forward the message to its superview. This allows any NSView to simply send the message to itself during a mouse-dragging loop without checking whether it's contained in an NSClipView (though it does need to check whether the mouse is outside of its visible portion, as returned by **visibleRect**).

Method Types

Calculating layout

+ contentSizeForFrameSize:hasHorizontalScroller:hasVerticalScroller:borderType:
+ frameSizeForContentSize:hasHorizontalScroller:hasVerticalScroller:borderType:

Determining component sizes

– contentSize
– documentVisibleRect

Managing graphic attributes

– setBackgroundColor:
– backgroundColor
– setBorderType:
– borderType

Managing the scrolled views

– setContentView:
– contentView
– setDocumentView:
– documentView
– setDocumentCursor:

Managing scrollers

– setHorizontalScroller:
– horizontalScroller
– setHasHorizontalScroller:
– hasHorizontalScroller
– setVerticalScroller:
– verticalScroller
– setHasVerticalScroller:
– hasVerticalScroller

Managing rulers

+ setRulerViewClass:
+ rulerViewClass
– setHasHorizontalRuler:
– hasHorizontalRuler
– setHorizontalRulerView:
– horizontalRulerView
– setHasVerticalRuler:
– hasVerticalRuler
– setVerticalRulerView:
– verticalRulerView
– setRulersVisible:
– rulersVisible
– isRulerVisible
– toggleRuler:

Setting scrolling behavior

– setLineScroll:
– lineScroll
– setPageScroll:
– pageScroll
– setScrollsDynamically:
– scrollsDynamically

Updating display after scrolling

– reflectScrolledClipView:

Arranging components

– tile

Class Methods

contentSizeForFrameSize:hasHorizontalScroller:hasVerticalScroller:borderType:

+ (NSSize)**contentSizeForFrameSize:**(NSSize)*frameSize*
hasHorizontalScroller:(BOOL)*hFlag*
hasVerticalScroller:(BOOL)*vFlag*
borderType:(NSBorderType)*borderType*

Returns the size of a content view for an NSScrollView whose frame size is *frameSize*. *hFlag* and *vFlag* indicate whether a horizontal or vertical scroller, respectively, is present. If the flag is YES then the content size is reduced in the appropriate dimension by the width of an NSScroller. *borderType* indicates the appearance of the NSScrollView's edge, which also affects the content size; see the description of **setBorderType:** for a list of possible values.

For an existing NSScrollView, you can simply use the **contentSize** method.

See also: + **frameSizeForContentSize:hasHorizontalScroller:hasVerticalScroller:borderType:**,
+ **scrollerWidth** (NSScroller)

frameSizeForContentSize:hasHorizontalScroller:hasVerticalScroller:borderType:

+ (NSSize)**frameSizeForContentSize:**(NSSize)*contentSize*
hasHorizontalScroller:(BOOL)*hFlag*
hasVerticalScroller:(BOOL)*vFlag*
borderType:(NSBorderType)*borderType*

Returns the frame size of an NSScrollView that contains a content view whose size is *contentSize*. *hFlag* and *vFlag* indicate whether a horizontal or vertical scroller, respectively, is present. If the flag is YES then the frame size is increased in the appropriate dimension by the width of an NSScroller. *borderType* indicates the appearance of the NSScrollView's edge, which also affects the frame size; see the description of **setBorderType:** for a list of possible values.

For an existing NSScrollView, you can simply use the **frame** method and extract its size.

See also: + **contentSizeForFrameSize:hasHorizontalScroller:hasVerticalScroller:borderType:**,
+ **scrollerWidth** (NSScroller)

rulerViewClass

+ (Class)**rulerViewClass**

Returns the default class to be used for ruler objects in NSScrollViews. This is normally NSRulerView.

See also: + **setRulerViewClass:**

setRulerViewClass:

+ (void)**setRulerViewClass:**(Class)*aClass*

Sets the default class to be used for ruler objects in NSScrollViews to *aClass*. This is normally NSRulerView, but you can use this method to set it to a custom subclass of NSRulerView.

Note: This method simply sets a global variable private to NSScrollView.Subclasses of NSScrollView should override both this method and **rulerViewClass** to store their ruler view classes in private variables.

See also: + **rulerViewClass**

Instance Methods

backgroundColor

– (NSColor *)**backgroundColor**

Returns the content view's background color.

See also: – **setBackgroundColor:**, – **backgroundColor** (NSClipView)

borderType

– (NSBorderType)**borderType**

Returns a value that represents the type of border surrounding the receiver; see the description of **setBorderType:** for a list of possible values.

contentSize

– (NSSize)**contentSize**

Returns the size of the receiver's content view.

See also: + **contentSizeForFrameSize:hasHorizontalScroller:hasVerticalScroller:borderType:**

contentView

– (NSClipView *)**contentView**

Returns the receiver's content view, the view that clips the document view.

See also: – **setContentView:**, – **documentView**

documentCursor

– (NSCursor *)**documentCursor**

Returns the content view's document cursor.

See also: – **setDocumentCursor:**, – **documentCursor** (NSClipView)

documentView

– (id)**documentView**

Returns the view that the receiver scrolls within its content view.

See also: – **setDocumentView:**, – **documentView** (NSClipView)

documentVisibleRect

– (NSRect)**documentVisibleRect**

Returns the portion of the document view, in its own coordinate system, that's visible through the receiver's content view.

See also: – **documentVisibleRect** (NSClipView), – **visibleRect** (NSView)

hasHorizontalRuler

– (BOOL)**hasHorizontalRuler**

Returns YES if the receiver maintains a horizontal ruler view, NO if it doesn't. Display of rulers is controlled using the **setRulersVisible:** method.

See also: – **horizontalRulerView**, – **setHasHorizontalRuler:**, – **hasVerticalRuler**, + **rulerViewClass**

hasHorizontalScroller

– (BOOL)**hasHorizontalScroller**

Returns YES if the receiver displays a horizontal scroller, NO if it doesn't.

See also: – **horizontalScroller**, – **setHasHorizontalScroller:**, – **hasVerticalScroller**

hasVerticalRuler

– (BOOL)**hasVerticalRuler**

Returns YES if the receiver maintains a vertical ruler view, NO if it doesn't. Display of rulers is controlled using the **setRulersVisible:** method.

See also: – **verticalRulerView**, – **setHasVerticalRuler:**, – **hasHorizontalRuler**, + **rulerViewClass**

hasVerticalScroller

– (BOOL)**hasVerticalScroller**

Returns YES if the receiver displays a vertical scroller, NO if it doesn't.

See also: – **verticalScroller**, – **setHasVerticalScroller:**, – **hasHorizontalScroller**

horizontalLineScroll

– (float)**horizontalLineScroll**

Returns the amount by which the receiver scrolls itself horizontally when scrolling line-by-line, expressed in the content view's coordinate system. This amount is used when the user clicks the scroll arrows on the horizontal scroll bar without holding a modifier key.

See also: – **setHorizontalLineScroll:,** – **verticalLineScroll,** – **setLineScroll:**, – **horizontalPageScroll**

horizontalPageScroll

– (float)**horizontalPageScroll**

Returns the amount of the document view kept visible when scrolling horizontally page-by-page, expressed in the content view's coordinate system. This amount is used when the user clicks the scroll arrows on the horizontal scroll bar while holding the Alternate key.

Note: This amount expresses the context that remains when the receiver scrolls by one page, allowing the user to orient himself to the new display. It differs from the line scroll amount, which indicates how far the document view moves. The page scroll amount is the amount *common to* the content view before and after the document view is scrolled by one page.

See also: – **setHorizontalPageScroll:,** – **verticalPageScroll,** – **setPageScroll:**, – **horizontalLineScroll**

horizontalRulerView

– (NSRulerView *)**horizontalRulerView**

Returns the receiver's horizontal ruler view, whether or not the receiver is currently displaying it, or **nil** if the receiver has none. If the receiver is set to display a horizontal ruler view and doesn't yet have one, this method creates an instance of the ruler view class set using the class method **setRulerViewClass:**. Display of rulers is controlled using the **setRulersVisible:** method.

See also: – **hasHorizontalRuler**, – **verticalRulerView**

horizontalScroller

– (NSScroller *)**horizontalScroller**

Returns the receiver's horizontal scroller, whether or not the receiver is currently displaying it, or **nil** if the receiver has none.

isRulerVisible

– (BOOL)**isRulerVisible**

This method is being deprecated in favor of **rulersVisible**. Use this method only when modifying older applications that use NSCStringText. New applications should use either NSText or NSTextView.

Returns YES if the receiver was set to show rulers using **setRulersVisible:** (whether or not it has rulers at all), NO if it was set to hide them.

See also: – **toggleRuler:**

lineScroll

– (float)**lineScroll**

Returns the vertical line scroll amount: the amount by which the receiver scrolls itself vertically when scrolling line-by-line, expressed in the content view's coordinate system. This amount is used when the user clicks the scroll arrows on the veritcal scroll bar without holding a modifier key. As part of its implementation, this method calls **verticalLineScroll**.

Note that a scroll view can have two different line scroll amounts: **veritcalLineScroll** and **horizontalLineScroll**. Use this method only if you can be sure they're both the same; for example, you always use **setLineScroll:**, which sets both amounts to the same value.

See also: – **setLineScroll:**, – **verticalPageScroll** , – **horizontalPageScroll**

pageScroll

– (float)**pageScroll**

Returns the vertical page scroll amount: the amount of the document view kept visible when scrolling vertically page-by-page, expressed in the content view's coordinate system. This amount is used when the user clicks the scroll arrows on the vertical scroll bar while holding the Alternate key. As part of its implementation, this method calls **verticalPageScroll**.

Note: This amount expresses the context that remains when the receiver scrolls by one page, allowing the user to orient himself to the new display. It differs from the line scroll amount, which indicates how

far the document view moves. The page scroll amount is the amount *common to* the content view before and after the document view is scrolled by one page.

Note that a scroll view can have two different page scroll amounts: **veritcalPageScroll** and **horizontalPageScroll**. Use this method only if you can be sure they're both the same; for example, you always use **setPageScroll:**, which sets both amounts to the same value.

See also: – **setPageScroll:**, – **verticalLineScroll**, – **horizontalLineScroll**

reflectScrolledClipView:

– (void)**reflectScrolledClipView:**(NSClipView *)*aClipView*

If *aClipView* is the receiver's content view, adjusts the receiver's scrollers to reflect the size and positioning of its document view. Does nothing if *aClipView* is any other view object (in particular, if it's an NSClipView that isn't the content view).

This method is invoked automatically during scrolling and when an NSClipView's relationship to its document view changes; you should rarely need to invoke it yourself, but may wish to override it for custom updating or other behavior.

See also: – **contentView**, – **documentView**

rulersVisible

– (BOOL)**rulersVisible**

Returns YES if the receiver was set to show rulers using **setRulersVisible:** (whether or not it has rulers at all), NO if it was set to hide them.

See also: – **hasHorizontalRuler**, – **hasVerticalRuler**

scrollsDynamically

– (BOOL)**scrollsDynamically**

Returns YES if the receiver redraws its document view while tracking the knob, NO if it redraws only when the scroller knob is released. NSScrollView scrolls dynamically by default.

See also: – **setScrollsDynamically:**

setBackgroundColor:

– (void)**setBackgroundColor:**(NSColor *)*aColor*

Sets the color of the content view's background to *aColor*. This color is used to paint areas inside the content view that aren't covered by the document view.

See also: – **backgroundColor**, – **setBackgroundColor:** (NSClipView)

setBorderType:

– (void)**setBorderType:**(NSBorderType)*borderType*

Sets the border type of the receiver to *borderType*, which may be one of:

NSNoBorder
NSLineBorder
NSBezelBorder
NSGrooveBorder

See also: – **borderType**

setContentView:

– (void)**setContentView:**(NSClipView *)*aView*

Sets the receiver's content view, the view that clips the document view, to *aView*. If *aView* has a document view, this method also sets the receiver's document view to be *aView*'s document view. The original content view retains its document view.

See also: – **contentView**, – **setDocumentView:**

setDocumentCursor:

– (void)**setDocumentCursor:**(NSCursor *)*aCursor*

Sets the cursor used when the mouse is over the content view to *aCursor*, by sending **setDocumentCursor:** to the content view.

See also: – **documentCursor**

setDocumentView:

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

Sets the receiver's document view to *aView*.

See also: – **documentView**, – **setDocumentView:** (NSClipView)

setHasHorizontalRuler:

– (void)**setHasHorizontalRuler:**(BOOL)*flag*

Determines whether the receiver keeps a horizontal ruler object. If *flag* is YES, the receiver allocates a horizontal ruler the first time it's needed. Display of rulers is handled independently with the **setRulersVisible:** method.

See also: – **hasHorizontalRuler**, – **horizontalRulerView**, – **setHasVerticalRuler:**

setHasHorizontalScroller:

– (void)**setHasHorizontalScroller:**(BOOL)*flag*

Determines whether the receiver keeps a horizontal scroller. If *flag* is YES, the receiver allocates and displays a horizontal scroller as needed. An NSScrollView by default has neither a horizontal nor a vertical scroller.

See also: – **hasHorizontalScroller**, – **horizontalScroller**, – **setHasVerticalScroller:**

setHasVerticalRuler:

– (void)**setHasVerticalRuler:**(BOOL)*flag*

Determines whether the receiver keeps a vertical ruler object. If *flag* is YES, the receiver allocates a vertical ruler the first time it's needed. Display of rulers is handled independently with the **setRulersVisible:** method.

See also: – **hasVerticalRuler**, – **verticalRulerView**, – **setHasHorizontalRuler:**, – **setRulersVisible:**

setHasVerticalScroller:

– (void)**setHasVerticalScroller:**(BOOL)*flag*

Determines whether the receiver keeps a vertical scroller. If *flag* is YES, the receiver allocates and displays a vertical scroller as needed. An NSScrollView by default has neither a vertical nor a horizontal scroller.

See also: – **hasVerticalScroller**, – **verticalScroller**, – **setHasHorizontalScroller:**

setHorizontalLineScroll:

– (void)**setHorizontalLineScroll:**(float)*aFloat*

Sets the amount by which the receiver scrolls itself horizontally when scrolling line-by-line to *aFloat*, expressed in the content view's coordinate system. This is the amount used when the user clicks the scroll arrows on the horizontal scroll bar without holding a modifier key. When displaying text in an NSScrollView, for example, you might set this to the height of a single line of text in the default font.

See also: – **lineScroll**, – **setPageScroll:**

setHorizontalPageScroll:

– (void)**setHorizontalPageScroll:**(float)*aFloat*

Sets the amount of the document view kept visible when scrolling horizontally page-by-page to *aFloat*, expressed in the content view's coordinate system. This amount is used when the user clicks the scroll arrows on the horizontal scroll bar while holding the Alternate key.

Note: This amount expresses the context that remains when the receiver scrolls by one page, allowing the user to orient himself to the new display. It differs from the line scroll amount, which indicates how far the document view moves. The page scroll amount is the amount *common to* the content view before and after the document view is scrolled by one page. Thus, setting the page scroll amount to 0.0 implies that the entire visible portion of the document view is replaced when a page scroll occurs.

See also: – **pageScroll**, – **setLineScroll:**

setHorizontalRulerView:

– (void)**setHorizontalRulerView:**(NSRulerView *)*aRulerView*

Sets the receiver's horizontal ruler view to *aRulerView*. You can use this method to override the default ruler class set using the class method **setRulerClass:**. Display of rulers is controlled using the **setRulersVisible:** method.

See also: – **horizontalRulerView**,– **setHasHorizontalRuler:**, – **setVerticalRulerView:**, – **setRulersVisible:**

setHorizontalScroller:

– (void)**setHorizontalScroller:**(NSScroller *)*aScroller*

Sets the receiver's horizontal scroller to *aScroller*, establishing the appropriate target-action relationships between them. To make sure the scroller is visible, invoke the **setHasHorizontalScroller:** method with an argument of YES.

See also: – **horizontalScroller**, – **setVerticalScroller:**

setLineScroll:

– (void)**setLineScroll:**(float)*aFloat*

Sets the horizontal and vertical line scroll amounts to *aFloat*. The line scroll is the amount by which the receiver scrolls itself when scrolling line-by-line, expressed in the content view's coordinate system. It's used when the user clicks the scroll arrows without holding a modifier key. When displaying text in an NSScrollView, for example, you might set this to the height of a single line of text in the default font.

As part of its implementation, this method calls **setVerticalLineScroll:** and **setHorizontalLineScroll:**.

See also: – **verticalLineScroll**, – **horizontalLineScroll**

setPageScroll:

– (void)**setPageScroll:**(float)*aFloat*

Sets the horizontal and vertical page scroll amounts to *aFloat*. The page scroll is the amount of the document view kept visible when scrolling page-by-page to *aFloat*, expressed in the content view's coordinate system. It's used when the user clicks the scroll arrows while holding the Alternate key.

Note: This amount expresses the context that remains when the receiver scrolls by one page, allowing the user to orient himself to the new display. It differs from the line scroll amount, which indicates how far the document view moves. The page scroll amount is the amount *common to* the content view before and after the document view is scrolled by one page. Thus, setting the page scroll amount to 0.0 implies that the entire visible portion of the document view is replaced when a page scroll occurs.

As part of its implementation, this method calls **setVerticalPageScroll:** and **setHorizontalPageScroll:**.

See also: – **verticalPageScroll**, – **verticalLineScroll**

setRulersVisible:

– (void)**setRulersVisible:**(BOOL)*flag*

Determines whether the receiver displays its rulers. If *flag* is YES, the receiver displays its rulers (creating them if needed). If *flag* is NO, the receiver doesn't display its rulers.

See also: – **rulersVisible**, – **hasHorizontalRuler**, – **hasVerticalRuler**

setScrollsDynamically:

– (void)**setScrollsDynamically:**(BOOL)*flag*

Determines whether the receiver redraws its document view while scrolling continuously. If *flag* is YES it does, if *flag* is NO it redraws only when the scroller knob is released. NSScrollView scrolls dynamically by default.

See also: – **scrollsDynamically**

setVerticalLineScroll:

– (void)**setVerticalLineScroll:**(float)*aFloat*

Sets the amount by which the receiver scrolls itself vertically when scrolling line-by-line to *aFloat*, expressed in the content view's coordinate system. This is the amount used when the user clicks the scroll arrows on the vertical scroll bar without holding a modifier key. When displaying text in an NSScrollView, for example, you might set this to the height of a single line of text in the default font.

See also: – **verticalLineScroll,** – **setHorizontalLineScroll:,** – **lineScroll**, – **setVerticalPageScroll:**

setVerticalPageScroll:

– (void)**setVerticalPageScroll:**(float)*aFloat*

Sets the amount of the document view kept visible when scrolling vertically page-by-page to *aFloat*, expressed in the content view's coordinate system. This amount is used when the user clicks the scroll arrows on the vertical scroll bar while holding the Alternate key.

Note: This amount expresses the context that remains when the receiver scrolls by one page, allowing the user to orient himself to the new display. It differs from the line scroll amount, which indicates how far the document view moves. The page scroll amount is the amount *common to* the content view before and after the document view is scrolled by one page. Thus, setting the page scroll amount to 0.0 implies that the entire visible portion of the document view is replaced when a page scroll occurs.

See also: – **verticalPageScroll,** – **setHorizontalPageScroll:,** – **pageScroll,** – **setVerticalLineScroll**

setVerticalRulerView:

– (void)**setVerticalRulerView:**(NSRulerView *)*aRulerView*

Sets the receiver's vertical ruler view to *aRulerView*. You can use this method to override the default ruler class set using the class method **setRulerClass:**. Display of rulers is controlled using the **setRulersVisible:** method.

See also: – **verticalRulerView**, – **setHasVerticalRuler:**, – **setHorizontalRulerView:**, – **setRulersVisible:**

setVerticalScroller:

– (void)**setVerticalScroller:**(NSScroller *)*aScroller*

Sets the receiver's vertical scroller to *aScroller*, establishing the appropriate target-action relationships between them. To make sure the scroller is visible, invoke the **setHasVerticalScroller:** method with an argument of YES.

See also: – **verticalScroller**, – **setHorizontalScroller:**

tile

– (void)**tile**

Lays out the components of the receiver: the content view, the scrollers, and the ruler views. You rarely need to invoke this method, but subclasses may override it to manage additional components.

toggleRuler:

– (void)**toggleRuler:**(id)*sender*

This method is being deprecated in favor of NSText's **toggleRuler:**. Use this method only when modifying older applications that use NSCStringText. New applications should use either NSText or NSTextView.

This action method shows or hides the ruler, if the receiver is enclosed in a scroll view.

See also: – **isRulerVisible**

verticalLineScroll

– (float)**verticalLineScroll**

Returns the amount by which the receiver scrolls itself vertically when scrolling line-by-line, expressed in the content view's coordinate system. This amount is used when the user clicks the scroll arrows on the vertical scroll bar without holding a modifier key.

See also: – **setVerticalLineScroll:**, – **horizontalLineScroll**, – **setLineScroll:**, – **verticalPageScroll**

verticalPageScroll

– (float)**verticalPageScroll**

Returns the amount of the document view kept visible when scrolling vertically page-by-page, expressed in the content view's coordinate system. This amount is used when the user clicks the scroll arrows on the vertical scroll bar while holding the Alternate key.

Note: This amount expresses the context that remains when the receiver scrolls by one page, allowing the user to orient himself to the new display. It differs from the line scroll amount, which indicates how far the document view moves. The page scroll amount is the amount *common to* the content view before and after the document view is scrolled by one page.

See also: – **setVerticalPageScroll:**, – **horizontalPageScroll**, – **setPageScroll:**, – **verticalLineScroll**

verticalRulerView

– (NSRulerView *)**verticalRulerView**

Returns the receiver's vertical ruler view, whether or not the receiver is currently displaying it, or **nil** if the receiver has none. If the receiver is set to display a vertical ruler view and doesn't yet have one, this method creates an instance of the ruler view class set using the class method **setRulerViewClass:**. Display of rulers is controlled using the **setRulersVisible:** method.

See also: – **hasVerticalRuler**, – **horizontalRulerView**

verticalScroller

– (NSScroller *)**verticalScroller**

Returns the receiver's vertical scroller, whether or not the receiver is currently displaying it, or **nil** if the receiver has none.

See also: – **hasVerticalScroller**, – **setVerticalScroller:**, – **horizontalScroller**

NSSecureTextField

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

Class Description

NSSecureTextField is a subclass of NSTextField that hides its text from display or other access via the user interface. It's suitable for use as a password-entry object, or for any item in which a secure value must be kept. An NSSecureTextField uses an NSSecureTextFieldCell, and adds behavior to the text system to protect its text value. Your code can get the text field's string value using the standard **stringValue** method, but users can't extract it themselves. NSSecureTextField overrides many aspects of text editing to prevent passing of the object's value out by mechanisms available to the user (namely, through Cut, Copy, and Paste commands, and the Services facility). This object also overrides the text system's drawing routine to draw no text at all.

NSSecureTextFieldCell

Inherits From: NSTextFieldCell : NSActionCell : NSCell : NSObject

Conforms To: NSCoding (NSCell)
NSCopying (NSCell)
NSObject (NSObject)

Declared In: AppKit/NSSecureTextField.h

Class Description

NSSecureTextFieldCell works with NSSecureTextField to provide a text field whose value is guarded from user examination. It overrides the general cell use of the field editor to provide its own field editor, which doesn't display text or allow the user to Cut, Copy, or Paste its value. See the NSSecureTextField class specification for more information.

NSSlider

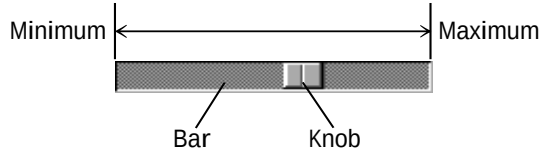
Inherits From: NSControl : NSView : NSResponder : NSObject

Conforms To: NSCoding (from NSResponder)
NSObject (from NSObject)

Declared In: AppKit/NSSlider.h

Class Description

An NSSlider displays a range of values for something in the application. An indicator, or knob, notes the current setting. The user can move the knob in the slider's bar to change the setting. If the slider is vertical, its minimum is at its bottom; if it is horizontal, its minimum is at its left.



The minimum and maximum can be obtained with the **minValue** and **maxValue** methods, and set with the **setMinValue:** and **setMaxValue:** methods. To read the value represented by the current position of the knob, you use an NSControl method like **floatValue**; conversely, to send a value to the slider, you use a NSControl method like **setFloatValue:**.

By default, an NSSlider is a continuous NSControl: while the user drags the slider's knob, the slider sends its action message continuously. If, instead, you want the slider to reserve its action message until the mouse is released, invoke **setContinuous:** (an NSControl method) with an argument of NO.

In its bar, an NSSlider can display an image, a title, or both. The title can be drawn in any color and any font. However, since a title in the bar may be obscured by the slider knob, you will more often label a slider by placing an NSTextField near it.

An NSSlider can have tick marks to either side of it. The values represented by the tick marks are determined by the number of tick marks and the minimum and maximum values of the slider; a slider's values can be pegged to the values represented by the tick marks.

Like most NSControls, NSSlider relies heavily on a related cell class, NSSliderCell. For more information, see the NSSliderCell class specification.

Don't use an NSSlider when an NSScroller would be better. A scroller represents the relative position of the visible portion of a view and lets the user choose which portion to view. A slider represents a range of values for something in the application and lets the user choose a setting.

Method Types

Asking about the slider's appearance

– altIncrementValue:
– image
– knobThickness
– isVertical

Changing the slider's appearance

– setAltIncrementValue:
– setImage:
– setKnobThickness:

Asking about the slider's title

– title
– titleCell
– titleColor
– titleFont

Changing the slider's title

– setTitle:
– setTitleCell:
– setTitleColor:
– setTitleFont:

Asking about the value limits

– maxValue
– minValue

Changing the value limits

– setMaxValue:
– setMinValue:

Handling mouse-down events

– acceptsFirstMouse:

Managing tick marks

– allowsTickMarkValuesOnly
– closestTickMarkValueToValue:
– indexOfTickMarkAtPoint:
– numberOfTickMarks
– rectOfTickMarkAtIndex:
– setAllowsTickMarkValuesOnly:
– setNumberOfTickMarks:
– setTickMarkPosition:
– tickMarkPosition
– tickMarkValueAtIndex:

Instance Methods

acceptsFirstMouse:

– (BOOL)**acceptsFirstMouse:**(NSEvent *)*mouseDownEvent*

Returns YES by default, so that a single mouse-down event can simultaneously activate the window and take hold of the slider's knob.

If you want the slider to wait for its own mouse-down event, you must override this method.

allowsTickMarkValuesOnly

– (BOOL)**allowsTickMarkValuesOnly**

Returns whether the receiver fixes its values to those values represented by its tick marks. In its implementation of this method, the receiving NSSlider simply invokes the method of the same name of its NSSliderCell.

See also: – **setAllowsTickMarkValuesOnly:**

altIncrementValue:

– (double)**altIncrementValue**

Returns the amount that the slider will change its value when the user drags the knob with the Alt key held down.

Unless you call **setAltIncrementValue:**, **altIncrementValue** returns –1.0, and the slider behaves no differently with the Alt key down than with it up.

See also: – **setAltIncrementValue:**

closestTickMarkValueToValue:

– (double)**closestTickMarkValueToValue**:(double)*aValue*;

Returns the value of the tick mark closest to *aValue*. In its implementation of this method, the receiving NSSlider simply invokes the method of the same name of its NSSliderCell.

See also: – **indexOfTickMarkAtPoint:**

image

– (NSImage *)**image**

Returns the image that the slider displays in its bar, or **nil** if no image has been set.

See also: – **setImage:**

indexOfTickMarkAtPoint:

– (int)**indexOfTickMarkAtPoint:**(NSPoint)*point*

Returns the index of the tick mark closest to the location of the slider represented by *point*. If *point* is not within the bounding rectangle (plus an extra pixel of space) of any tick mark, the method returns NSNotFound. In its implementation of this method, the receiving NSSlider simply invokes the method of the same name of its NSSliderCell. This method invokes **rectOfTickMarkAtIndex:** for each tick mark on the slider until it finds a tick mark containing the point.

See also: – **closestTickMarkValueToValue:**

isVertical

– (int)**isVertical**

Returns 1 if the slider is vertical, 0 if it's horizontal, and –1 if the orientation can't be determined (for example, if the slider hasn't been displayed yet). A slider is defined as vertical if its height is greater than its width.

knobThickness

– (float)**knobThickness**

Returns the knob's thickness, in pixels. The thickness is defined to be the extent of the knob along the long dimension of the bar. In a vertical slider, then, a knob's thickness is its height; in a horizontal slider, a knob's thickness is its width.

See also: – **setKnobThickness:**

maxValue

– (double)**maxValue**

Returns the maximum value that the slider can send to its target. A horizontal slider sends its maximum value when the knob is at the right end of the bar; a vertical slider sends it when the knob is at the top.

See also: – **setMaxValue:**

minValue

– (double)**minValue**

Returns the minimum value that the slider can send to its target. A vertical slider sends its minimum value when its knob is at the bottom; a horizontal slider, when its knob is all the way to the left.

See also: – **setMinValue:**

numberOfTickMarks

– (int)**numberOfTickMarks**

Returns the number of tick marks associated with the slider. The tick marks assigned to the minimum and maximum values are included. In its implementation of this method, the receiving NSSlider simply invokes the method of the same name of its NSSliderCell.

See also: – **setNumberOfTickMarks:**

rectOfTickMarkAtIndex:

– (NSRect)**rectOfTickMarkAtIndex:**(int)*index*

Returns the bounding rectangle of the tick mark identified by *index* (the minimum-value tick mark is at index 0). If no tick mark is associated with *index*, the method raises NSRangeException. In its

implementation of this method, the receiving NSSlider simply invokes the method of the same name of its NSSliderCell.

See also: – **indexOfTickMarkAtPoint:**

setAllowsTickMarkValuesOnly:

– (void)**setAllowsTickMarkValuesOnly:**(BOOL)*flag*

Sets whether the receiver's values are fixed to the values represented by the tick marks. For example, if a slider has a minimum value of 0, a maximum value of 100, and five markers, the allowable values are 0, 25, 50, 75, and 100. When users move the slider's knob, it jumps to the tick mark nearest the cursor when the mouse is released. This method has no effect if the slider has no tick marks. In its implementation of this method, the receiving NSSlider simply invokes the method of the same name of its NSSliderCell.

See also: – **allowsTickMarkValuesOnly**

setAltIncrementValue:

– (void)**setAltIncrementValue:**(double)*increment*

Sets the amount by which the NSSliderCell modifies its value when the user Alt-drags the knob. *increment* must fit the range of values that the slider can represent—for example, if the slider has a minimum value of 5 and a maximum value of 10, *increment* should be between 0 and 5. If *increment* is outside that range, the value is unchanged.

If you don't call this method, the slider behaves the same with the Alt key down as with it up. This is also the result when you call **setAltIncrementValue:** with an *increment* of -1.

See also: – **maxValue**, – **minValue**

setImage:

– (void)**setImage:**(NSImage *)*barImage*

Sets the image that the slider displays in the bar behind its knob. The slider may scale and distort *barImage* to fit inside the bar.

The knob may cover part of the image. If you want the image to be visible all the time, you're better off placing it near the slider.

See also: – **setImage:**

setKnobThickness:

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

Lets you set the knob's thickness, measured in pixels. The thickness is defined to be the extent of the knob along the long dimension of the bar. In a vertical slider, then, a knob's thickness is its height; in a horizontal slider, a knob's thickness is its width.

See also: – **knobThickness**

setMaxValue:

– (void)**setMaxValue:**(double)*maxValue*

Sets the maximum value that the slider can send to its target—the value that a horizontal slider sends when its knob is all the way to the right, or that a vertical slider sends when its knob is at the top.

See also: – **maxValue**

setMinValue:

– (void)**setMinValue:**(double)*minValue*

Sets the minimum value that the slider can send to its target. A horizontal slider sends its minimum value when its knob is all the way to the left; a vertical slider sends its minimum value when its knob is at the bottom.

See also: – **minValue**

setNumberOfTickMarks:

– (void)**setNumberOfTickMarks:**(int)*numberOfTickMarks*

Sets the number of tick marks displayed by the receiver (which include those assigned to the minimum and maximumvalues). By default, this value is zero, and no tick marks appear. The number of tick marks assigned to a slider, along with the slider's minimum and maximum values, determine the values associated with the tick marks. In its implementation of this method, the receiving NSSlider simply invokes the method of the same name of its NSSliderCell.

See also: – **numberOfTickMarks**

setTickMarkPosition:

– (void)**setTickMarkPosition:**(NSTickMarkPosition)*position*

Sets where tick marks appear relative to the receiver. For horizontal sliders, *position* can be NSTickMarksBelow (the default) or NSTickMarksAbove; for vertical sliders, *position* can be NSTickMarksLeft (the default) or NSTickMarksRight. This method has no effect if no tick marks have been assigned (that is, **numberOfTickMarks** returns zero). In its implementation of this method, the receiving NSSlider simply invokes the method of the same name of its NSSliderCell.

See also: – **tickMarkPosition**

setTitle:

– (void)**setTitle:**(NSString *)*barTitle*

Sets the title that the slider displays in the bar behind its knob.

The knob may cover part or all of the title. If you want the title to be visible all of the time, you're better off placing a label near the slider.

See also: – **title**

setTitleCell:

– (void)**setTitleCell:**(NSCell *)*titleCell*

Sets the cell used to draw the slider's title. You only need to invoke this method if the default title cell, NSTextFieldCell, doesn't suit your needs—that is, if you want to display the title in a manner that NSTextFieldCell doesn't permit. When you do choose to override the default, *titleCell* should be an instance of a subclass of TextFieldCell.

See also: – **titleCell**

setTitleColor:

– (void)**setTitleColor:**(NSColor *)*color*

Sets the color used to draw the slider's title.

See also: – **titleColor**

setTitleFont:

– (void)**setTitleFont:**(NSFont *)*font*

Sets the font used to draw the slider's title.

See also: – **titleFont**

– tickMarkPosition

– (NSTickMarkPosition)**tickMarkPosition**

Returns how the receiver's tick marks are aligned with it: NSTickMarkBelow, NSTickMarkAbove, NSTickMarkLeft, or NSTickMarkRight (the last two are for vertical sliders). The default alignments are NSTickMarkBelow and NSTickMarkLeft. In its implementation of this method, the receiving NSSlider simply invokes the method of the same name of its NSSliderCell.

See also: – **setTickMarkPosition:**

– tickMarkValueAtIndex:

– (double)**tickMarkValueAtIndex:**(int)*index*

Returns the receiver's value represented by the tick mark at index (the minimum-value tick mark has an index of zero). In its implementation of this method, the receiving NSSlider simply invokes the method of the same name of its NSSliderCell.

title

– (NSString *)**title**

Returns the slider's title. The default title is the empty string ("").

See also: – **setTitle:**

titleCell

– (id)**titleCell**

Returns the cell used to draw the title. The default is an NSTextFieldCell.

See also: – **setTitleCell:**

titleColor

– (NSColor *)**titleColor**

Returns the color used to draw the slider's title. The default color is NSColor's **controlTextColor**.

See also: – **setTitleColor:**

titleFont

– (NSFont *)**titleFont**

Returns the font used to draw the slider's title.

See also: – **setTitleColor:**

NSSliderCell

Inherits From: NSActionCell : NSCell : NSObject

Conforms To: NSCoding, NSCopying (from NSCell)
NSObject (from NSObject)

Declared In: AppKit/NSSliderCell.h

Class Description

An NSSliderCell controls the appearance and behavior of an NSSlider, or of a single slider in an NSMatrix. A slider can have tick marks to either side of it. The values represented by the tick marks are determined by the number of tick marks and the minimum and maximum values of the slider; a slider's values can be pegged to the values represented by the tick marks.

You can customize an NSSliderCell to a certain degree, using its **set...** methods. If these do not allow you sufficient flexibility, you can create a subclass. In that subclass, you can override any of the following methods: **knobRectFlipped:**, **drawBarInside:flipped:**, **drawKnob:**, or **prefersTrackingUntilMouseUp**.

Method Types

Asking about the cell's behavior

– altIncrementValue
+ prefersTrackingUntilMouseUp
– trackRect

Changing the cell's behavior

– setAltIncrementValue:

Displaying the cell

– knobRectFlipped:
– drawBarInside:flipped:
– drawKnob
– drawKnob:

Asking about the cell's appearance

– knobThickness
– isVertical
– title
– titleCell
– titleFont
– titleColor

Changing the cell's appearance

– setKnobThickness:
– setTitle:
– setTitleCell:
– setTitleColor:
– setTitleFont:

Asking about the value limits

– maxValue
– minValue

Changing the value limits

– setMaxValue:
– setMinValue:

Managing tick marks

– allowsTickMarkValuesOnly
– closestTickMarkValueToValue:
– indexOfTickMarkAtPoint:
– numberOfTickMarks
– rectOfTickMarkAtIndex:
– setAllowsTickMarkValuesOnly:
– setNumberOfTickMarks:
– setTickMarkPosition:
– tickMarkPosition
– tickMarkValueAtIndex:

Class Methods

prefersTrackingUntilMouseUp

+ (BOOL)**prefersTrackingUntilMouseUp**

By default, this method returns YES, so that an NSSliderCell continues to track the mouse even after the mouse leaves the cell's tracking rectangle. This means that, once you take hold of a slider's knob (by putting the mouse inside the cell's frame rectangle and pressing the mouse button), you retain control of the knob until you release the mouse button, even if you drag the mouse clear to the other side of the screen.

Never call this method explicitly. Override it if you create a subclass of NSSliderCell that you want to track the mouse differently.

Instance Methods

allowsTickMarkValuesOnly

– (BOOL)**allowsTickMarkValuesOnly**

Returns whether the receiver fixes its values to those values represented by its tick marks.

See also: – **setAllowsTickMarkValuesOnly:**

altIncrementValue

– (double)**altIncrementValue**

Returns the amount that the slider will change its value when the user drags the knob with the Alt key held down.

Unless you call **setAltIncrementValue**, **altIncrementValue** returns –1.0, and the slider behaves no differently with the Alt key down than with it up.

See also: – **setAltIncrementValue:**

closestTickMarkValueToValue:

– (double)**closestTickMarkValueToValue**:(double)*aValue*;

Returns the value of the tick mark closest to *aValue*.

See also: – **indexOfTickMarkAtPoint:**

drawBarInside:flipped:

– (void)**drawBarInside:**(NSRect)*aRect* **flipped:**(BOOL)*flipped*

Draws the slider's bar—but not its bezel or knob—in *aRect*.

flipped indicates whether the cell's control view—that is, the NSSlider or NSMatrix associated with the NSSliderCell—has a flipped coordinate system.

You should never invoke this method explicitly. It's included so that you can override it in a subclass.

See also: – **drawKnob:**

drawKnob

– (void)**drawKnob**

Calculates the rectangle in which the knob should be drawn, then invokes **drawKnob:** to actually draw the knob. Before this message is sent, a **lockFocus** method must be sent to the cell's control view.

You might invoke this method if you override one of the display methods belonging to NSControl or NSCell.

If you create a subclass of NSSliderCell, don't override this method. Override **drawKnob:** instead.

drawKnob:

– (void)**drawKnob:**(NSRect)*knobRect*

Draws the knob in *knobRect*. Before this message is sent, a **lockFocus** message must be sent to the cell's control view.

You should never invoke this method explicitly. It's included so that you can override it in a subclass.

indexOfTickMarkAtPoint:

– (int)**indexOfTickMarkAtPoint:**(NSPoint)*point*

Returns the index of the tick mark closest to the location of the slider represented by *point*. If *point* is not within the bounding rectangle (plus an extra pixel of space) of any tick mark, the method returns NSNotFound. This method invokes **rectOfTickMarkAtIndex:** for each tick mark on the slider until it finds a tick mark containing the point.

isVertical

– (int)**isVertical**

Returns 1 if the slider is vertical, 0 if it's horizontal, and –1 if the orientation can't be determined (for example, if the slider hasn't been displayed yet). A slider is defined as vertical if its height is greater than its width.

knobRectFlipped:

– (NSRect)**knobRectFlipped:**(BOOL)*flipped*

Returns the rectangle in which the knob will be drawn, specified in the coordinate system of the NSSlider or NSMatrix with which the NSSliderCell is associated. *flipped* indicates whether that coordinate system is flipped, a question you can answer by sending NSView's **isFlipped** message to the NSMatrix or NSSlider.

The knob rectangle depends on where in the slider the knob belongs—that is, it depends on the SliderCell's minimum and maximum values, and on the value which the position of the knob will represent.

You should never invoke this method explicitly. It's included so that you can override it in a subclass.

knobThickness

– (float)**knobThickness**

Returns the knob's thickness, in pixels. The thickness is defined to be the extent of the knob along the long dimension of the bar. In a vertical slider, then, a knob's thickness is its height; in a horizontal slider, its thickness is its width.

See also: – **setKnobThickness:**

maxValue

– (double)**maxValue**

Returns the maximum value that the slider can send to its target. A horizontal slider sends its maximum value when the knob is at the right end of the slider; a vertical slider sends it when the knob is at the top.

See also: – **setMaxValue:**

minValue

– (double)**minValue**

Returns the minimum value that the slider can send to its target. A vertical slider sends this value when its knob is at the bottom; a horizontal slider sends it when its knob is all the way to the left.

numberOfTickMarks

– (int)**numberOfTickMarks**

Returns the number of tick marks associated with the slider. The tick marks assigned to the minimum and maximum values are included.

See also: – **setNumberOfTickMarks:**

rectOfTickMarkAtIndex:

– (NSRect)**rectOfTickMarkAtIndex:**(int)*index*

Returns the bounding rectangle of the tick mark identified by *index* (the minimum-value tick mark is at index 0). If no tick mark is associated with *index*, the method raises NSRangeException.

See also: – **indexOfTickMarkAtPoint:**

setAllowsTickMarkValuesOnly:

– (void)**setAllowsTickMarkValuesOnly:**(BOOL)*flag*

Sets whether the receiver's values are fixed to the values represented by the tick marks. For example, if a slider has a minimum value of 0, a maximum value of 100, and five markers, the allowable values are 0, 25, 50, 75, and 100. When users move the slider's knob, it jumps to the tick mark nearest the cursor when the mouse is released. This method has no effect if the slider has no tick marks.

See also: – **allowsTickMarkValuesOnly**

setAltIncrementValue:

– (void)**setAltIncrementValue:**(double)*increment*

Sets the amount by which the NSSliderCell modifies its value when the knob is Alt-dragged. *increment* should fit the range of values that the slider can represent—for example, if the slider has a minimum value of 5 and a maximum value of 10, *increment* should be between 0 and 5.

If you don't call this method, the slider behaves the same with the Alt key down as with it up. This is also the result when you call **setAltIncrementValue:** with an *increment* of -1.

See also: – **maxValue**, – **minValue**

setKnobThickness:

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

Lets you set the knob's thickness, measured in pixels. The thickness is defined to be the extent of the knob along the long dimension of the bar. In a vertical slider, then, a knob's thickness is its height; in a horizontal slider, its thickness is its width.

See also: – **knobThickness**

setMaxValue:

– (void)**setMaxValue:**(double)*aDouble*

Sets the maximum value that the slider can send to its target—the value that a horizontal slider will send when its knob is all the way to the right, or that a vertical slider will send when its knob is at the top.

See also: – **maxValue**

setMinValue:

– (void)**setMinValue:**(double)*aDouble*

Sets the minimum value that the slider can send to its target. A horizontal slider sends its minimum value when its knob is all the way to the left; a vertical slider sends its minimum value when its knob is at the bottom.

See also: – **minValue**

setNumberOfTickMarks:

– (void)**setNumberOfTickMarks:**(int)*numberOfTickMarks*

Sets the number of tick marks displayed by the receiver (which include those assigned to the minimum and maximumvalues). By default, this value is zero, and no tick marks appear. The number of tick marks assigned to a slider, along with the slider's minimum and maximum values, determine the values associated with the tick marks.

See also: – **numberOfTickMarks**

setTickMarkPosition:

– (void)**setTickMarkPosition:**(NSTickMarkPosition)*position*

Sets where tick marks appear relative to the receiver. For horizontal sliders, *position* can be NSTickMarksBelow (the default) or NSTickMarksAbove; for vertical sliders, *position* can be NSTickMarksLeft (the default) or NSTickMarksRight. This method has no effect if no tick marks have been assigned (that is, **numberOfTickMarks** returns zero).

See also: – **tickMarkPosition**

setTitle:

– (void)**setTitle:**(NSString *)*title*

Sets the title in the bar behind the slider's knob to *title*.

See also: – **title**

setTitleCell:

– (void)**setTitleCell:**(NSCell *)*aCell*

Sets the cell used to draw the slider's title. You only need to invoke this method if the default title cell, NSTextFieldCell, doesn't suit your needs—that is, if you want to display the title in a manner that NSTextFieldCell doesn't permit. When you do choose to override the default, *aCell* should be an instance of a subclass of TextFieldCell.

See also: – **titleCell**

setTitleColor:

– (void)**setTitleColor:**(NSColor *)*color*

Sets the color used to draw the slider's title.

See also: – **titleColor**

setTitleFont:

– (void)**setTitleFont:**(NSFont *)*font*

Sets the font used to draw the slider's title.

See also: – **titleFont**

– tickMarkPosition

– (NSTickMarkPosition)**tickMarkPosition**

Returns how the receiver's tick marks are aligned with it: NSTickMarkBelow, NSTickMarkAbove, NSTickMarkLeft, or NSTickMarkRight (the last two are for vertical sliders). The default alignments are NSTickMarkBelow and NSTickMarkLeft.

See also: – **setTickMarkPosition:**

– tickMarkValueAtIndex:

– (double)**tickMarkValueAtIndex:**(int)*index*

Returns the receiver's value represented by the tick mark at index (the minimum-value tick mark has an index of zero).

title

– (NSString *)**title**

Returns the slider's title. The default title is the empty string ("").

See also: – **setTitle:**

titleCell

– (id)**titleCell**

Returns the cell used to draw the title. The default is an NSTextFieldCell.

See also: – **setTitleCell:**

titleColor

– (NSColor *)**titleColor**

Returns the color used to draw the slider's title. The default color is NSColor's **controlTextColor**.

See also: – **setTitleColor:**

titleFont

– (NSFont *)**titleFont**

Returns the font used to draw the slider's title.

See also: – **setTitleFont:**

trackRect

– (NSRect)**trackRect**

Returns the rectangle within which the cell tracks the mouse while the mouse button is down. This rectangle includes the slider bar, but not the bezel.

NSSpellChecker

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: AppKit/NSSpellChecker.h

Class Description

The NSSpellChecker class gives any application an interface to the OpenStep spell-checking service. To handle all its spell checking, an application needs only one instance of NSSpellChecker. It provides a panel in which the user can specify decisions about words that are suspect. To check the spelling of a piece of text, the application:

- Includes in its user interface a menu item (or a button or command) by which the user will request spell checking.
- Makes the text available by way of an NSString object.
- Creates an instance of the NSSpellChecker class and sends it a **checkSpellingOfString:startingAt:** message.

For example, you might use the following statement to create a spell checker:

```
range = [[NSSpellChecker sharedSpellChecker] checkSpellingOfString:aString startingAt:0];
```

The **checkSpellingOfString:startingAt:** method checks the spelling of the words in the specified string beginning at the specified offset (this example uses 0 to start at the beginning of the string) until it finds a word that is misspelled. Then it returns an NSRange to indicate the location of the misspelled word.

In a graphical application, whenever a misspelled word is found, you'll probably want to highlight the word in the document, using the NSRange that **checkSpellingOfString:startingAt:** returned to determine the text to highlight. Then you should show the misspelled word in the Spelling panel's misspelled-word field by calling **updateSpellingPanelWithMisspelledWord:**. If **checkSpellingOfString:startingAt:** does not find a misspelled word, you should call **updateSpellingPanelWithMisspelledWord:** with the empty string. This causes the system to beep, letting the user know that the spell check is complete and no misspelled words were found. None of these steps is required, but if you do one, you should do them all.

The object that provides the string being checked should adopt the following protocols:

Protocol	Description
NSChangeSpelling	A message in this protocol (**changeSpelling:**) is sent down the responder chain when the user presses the Correct button.
NSIgnoreMisspelledWords	When the object being checked responds to this protocol, the spell server keeps a list of words that are acceptable in the document and enables the Ignore button in the Spelling panel.

The application may choose to split a document's text into segments and check them separately. This will be necessary when the text has segments in different languages. Spell checking is invoked for one language at a time, so a document that contains portions in three languages will require at least three checks.

Dictionaries and Word Lists

The process of checking spelling makes use of three references:

- A dictionary registered with the system's spell-checking service. When the Spelling panel first appears, by default it shows the dictionary for the user's preferred language. The user may select a different dictionary from the list in the Spelling panel.
- The user's "learn" list of correctly-spelled words in the current language. The NSSpellChecker updates the list when the user presses the Learn or Forget buttons in the Spelling panel.
- The document's list of words to be ignored while checking it (if the first responder conforms to the NSIgnoreMisspelledWords protocol). The NSSpellChecker updates its copy of this list when the user presses the Ignore button in the Spelling panel.

A word is considered to be misspelled if none of these three accepts it.

Matching a List of Ignored Words with the Document It Belongs To

The NSString being checked isn't the same as the document. In the course of processing a document, an application might run several checks based on different parts or different versions of the text. But they'd all belong to the same document. The NSSpellChecker keeps a separate "ignored words" list for each document that it checks. To help match "ignored words" lists to documents, you should call **uniqueSpellDocumentTag** once for each document. This method returns a unique arbitrary integer that will serve to distinguish one document from the others being checked and to match each "ignored words" list to a document. When searching for misspelled words, pass the tag as the fourth argument of **checkSpellingOfString:startingAt:language:wrap:inSpellDocumentWithTag:wordCount:**. (The convenience method **checkSpellingOfString:startingAt:** takes no tag. This method is suitable when the first responder does not conform to the NSIgnoreMisspelledWords protocol.)

When the application saves a document, it may choose to retrieve the “ignored words” list and save it along with the document. To get back the right list, it must send the NSSpellChecker an **ignoredWordsInSpellDocumentWithTag:** message. When the application has closed a document, it should notify the NSSpellChecker that the document’s “ignored words” list can now be discarded, by sending it a **closeSpellDocumentWithTag:** message. When the application reopens the document, it should restore the “ignored words” list with the message **setIgnoredWords:inSpellDocumentWithTag:**.

Method Types

Getting the spell checker

+ sharedSpellChecker
+ sharedSpellCheckerExists

Managing the spelling panel

– setAccessoryView:
– accessoryView
– spellingPanel

Checking spelling

– countWordsInString:language:
– checkSpellingOfString:startingAt:
– checkSpellingOfString:startingAt:language:wrap:
inSpellDocumentWithTag:wordCount:

Setting the language

– setLanguage:
– language

Managing the Spelling Process

+ uniqueSpellDocumentTag
– closeSpellDocumentWithTag:
– ignoreWord:inSpellDocumentWithTag:
– setIgnoredWords:inSpellDocumentWithTag:
– ignoredWordsInSpellDocumentWithTag:
– setWordFieldStringValue:
– updateSpellingPanelWithMisspelledWord:

Class Methods

sharedSpellChecker

+ (NSSpellChecker *)**sharedSpellChecker**

Returns the NSSpellChecker (one per application).

See also: + **sharedSpellCheckerExists**

sharedSpellCheckerExists

+ (BOOL)**sharedSpellCheckerExists**

Returns whether the application's NSSpellChecker has already been created.

See also: + **sharedSpellChecker**

uniqueSpellDocumentTag

+ (int)**uniqueSpellDocumentTag**

Returns a guaranteed unique tag to use as the spell-document tag for a document. Use this method to generate tags to avoid collisions with other objects that can be spell-checked.

Instance Methods

accessoryView

– (NSView *)**accessoryView**

Returns the Spelling panel's accessory NSView object.

See also: – **setAccessoryView:**

checkSpellingOfString:startingAt:

– (NSRange)**checkSpellingOfString:**(NSString *)*stringToCheck*
startingAt:(int)*startingOffset*

Starts the search for a misspelled word in *stringToCheck* starting at *startingOffset* within the string object. Returns the range of the first misspelled word. Wrapping occurs but no ignored-words dictionary is used.

checkSpellingOfString:startingAt:language:wrap:inSpellDocumentWithTag: wordCount:

– (NSRange)**checkSpellingOfString:**(NSString *)*stringToCheck*
startingAt:(int)*startingOffset*
language:(NSString *)*language*
wrap:(BOOL)*wrapFlag*
inSpellDocumentWithTag:(int)*tag*
wordCount:(int *)*wordCount*

Starts the search for a misspelled word in *stringToCheck* starting at *startingOffset* within the string object. Returns the range of the first misspelled word and optionally the word count by reference. *tag* is an identifier unique within the application used to inform the spell check which document (actually, a dictionary) of ignored words to use. *wrapFlag* determines whether spell checking continues at the beginning of the string when the end is reached. *language* is the language used in the string. If *language* is the empty string, the current selection in the Spelling panel's pop-up menu is used.

closeSpellDocumentWithTag:

– (void)**closeSpellDocumentWithTag:**(int)*tag*

Notifies the spell checker that the user has finished with the ignored-word document identified by *tag*, causing it to throw that dictionary away.

countWordsInString:language:

– (int)**countWordsInString:**(NSString *)*stringToCount*
language:(NSString *)*language*

Returns the number of words in *stringToCount*. The *language* argument specifies the language used in the string. If *language* is the empty string, the current selection in the Spelling panel's pop-up menu is used.

ignoreWord:inSpellDocumentWithTag:

– (void)**ignoreWord:**(NSString *)*wordToIgnore* **inSpellDocumentWithTag:**(int)*tag*

Instructs the spell checker to ignore all future occurrences of *wordToIgnore* in the document identified by *tag*. You should invoke this method from within your implementation of the NSIgnoreMisspelledWords protocol's **ignoreSpelling:** method.

ignoredWordsInSpellDocumentWithTag:

– (NSArray *)**ignoredWordsInSpellDocumentWithTag:**(int)*tag*

Returns the array of ignored words for a document identified by *tag*. Invoke this before **closeSpellDocumentWithTag:** if you want to store the ignored words.

See also: – **setIgnoredWords:inSpellDocumentWithTag:**

language

– (NSString *)**language**

Returns the current language used in spell-checking.

See also: – **setLanguage:**

setAccessoryView:

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

Makes an NSView object an accessory of the Spelling panel by making it a subview of the panel's content view. This method posts the notification NSWindowDidResizeNotification with the Spelling panel object to the default notification center.

See also: – **accessoryView**

setIgnoredWords:inSpellDocumentWithTag:

– (void)**setIgnoredWords:**(NSArray *)*someWords* **inSpellDocumentWithTag:**(int)*tag*

Initializes the ignored-words document (i.e., dictionary identified by *tag* with *someWords)*, an array of words to ignore.

See also: – **ignoredWordsInSpellDocumentWithTag:**

setLanguage:

– (BOOL)**setLanguage:**(NSString *)*language*

Sets the language to use in spell-checking to *language*. Returns whether the Language pop-up list in the Spelling panel lists *language*.

See also: – **language**

setWordFieldStringValue:

– (void)**setWordFieldStringValue:**(NSString *)*aString*

Sets the string that appears in the misspelled word field, using the string object *aString*.

spellingPanel

– (NSPanel *)**spellingPanel**

Returns the spell checker's panel.

updateSpellingPanelWithMisspelledWord:

– (void)**updateSpellingPanelWithMisspelledWord:**(NSString *)*word*

Causes the spell checker to update the Spelling panel's misspelled-word field to reflect *word*. You are responsible for highlighting *word* in the document and for extracting it from the document using the range returned by the **checkSpelling:...** methods. Pass the empty string as *word* to have the system beep, indicating no misspelled words were found.

NSSpellServer

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: AppKit/NSSpellServer.h

Class Description

The NSSpellServer class gives you a way to make your particular spelling checker a service that's available to any application. A *service* is an application that declares its availability in a standard way, so that any other applications that wish to use it can do so. If you build a spelling checker that makes use of the NSSpellServer class and list it as an available service, then users of any application that makes use of NSSpellChecker or includes a Services menu will see your spelling checker as one of the available dictionaries.

To make use of NSSpellServer, you write a small program that creates an NSSpellServer instance and a delegate that responds to messages asking it to find a misspelled word and to suggest guesses for a misspelled word. Send the NSSpellServer **registerLanguage:byVendor:** messages to tell it the languages your delegate can handle.

The program that runs your spelling checker should not be built as an Application Kit application, but as a simple program. Suppose you supply spelling checkers under the vendor name "Acme." Suppose the file containing the code for your delegate is called AcmeEnglishSpellChecker. Then the following might be your program's **main**:

```
void main()
{
    NSSpellServer *aServer = [[NSSpellServer alloc] init];
    if ([aServer registerLanguage:"English" byVendor:"Acme"]) {
        [aServer setDelegate:[[AcmeEnglishSpellChecker alloc] init]];
        [aServer run];
        fprintf(stderr, "Unexpected death of Acme SpellChecker!\n");
    else {
        fprintf(stderr, "Unable to check in Acme SpellChecker.\n");
    }
}
```

Your delegate is an instance of a custom subclass. (It's simplest to make it a subclass of NSObject, but that's not a requirement.) Given an NSString, your delegate must be able to find a misspelled word by implementing the method **spellServer:findMisspelledWordInString:language:wordCount:countOnly:** . Usually, this method also reports the number of words it has scanned, but that isn't mandatory.

Optionally, the delegate may also suggest corrections for misspelled words. It does so by implementing the method **spellServer:suggestGuessesForWord:inLanguage:**.

Service Availability Notice

When there's more than one spelling checker available, the user selects the one desired. The application that requests a spelling check uses an NSSpellChecker object, and it provides a Spelling panel; in the panel there's a pop-up list of available spelling checkers. Your spelling checker appears in that list if it has a *service descriptor.*

A service descriptor is an entry in a text file called **services**. Usually it's located within the bundle that also contains your spelling checker's executable file. The bundle (or directory) that contains the services file must have a name ending in ".service" or ".app". The system looks for service bundles in a standard set of directories.

A spell checker service availability notice has a standard format, illustrated in the following example for the Acme spelling checker:

```
Spell Checker:  Acme
Language:  French
Language:  English
Executable:  franglais.daemon
```

The first line identifies the type of service; for a spelling checker, it must say "Spell Checker:" followed by your vendor name. The next line contains the English name of a language your spelling checker is prepared to check. (The language must be one your system recognizes.) If your program can check more than one language, use an additional line for each additional language. The last line of a descriptor gives the name of the service's executable file. (It requires a complete path if it's in a different directory.)

If there's a service descriptor for your Acme spelling checker and also a service descriptor for the English checker provided by a vendor named Consolidated, a user looking at the Spelling panel's pop-up list would see:

```
English (Acme)
English (Consolidated)
French (Acme)
```

Illustrative Sequence of Messages to an NSSpellServer

The act of checking spelling usually involves the interplay of objects in two classes: the user application's NSSpellChecker (which responds to interactions with the user) and your spelling checker's NSSpellServer (which provides the application interface for your spelling checker). You can see the interaction between the two in the following list of steps involved in finding a misspelled word.

- The user of an application selects a menu item to request a spelling check. The application sends a message to its NSSpellChecker object. The NSSpellChecker in turn sends a corresponding message to the appropriate NSSpellServer.

- The NSSpellServer receives the message asking it to check the spelling of an NSString. It forwards the message to its delegate.
- The delegate searches for a misspelled word. If it finds one, it returns an NSRange identifying the word's location in the string.
- The NSSpellServer receives a message asking it to suggest guesses for the correct spelling of a misspelled word, and forwards the message to its delegate.
- The delegate returns a list of possible corrections, which the NSSpellServer in turn returns to the NSSpellChecker that initiated the request.
- The NSSpellServer doesn't know what the user does with the errors its delegate has found or with the guesses its delegate has proposed. (Perhaps the user corrects the document, perhaps by selecting a correction from the NSSpellChecker's display of guesses; but that's not the NSSpellServer's responsibility.) However, if the user presses the Learn or Forget buttons (thereby causing the NSSpellChecker to revise the user's word list), the NSSpellServer receives a notification of the word thus learned or forgotten. It's up to you whether your spell checker acts on this information. If the user presses the Ignore button, the delegate is not notified (but the next time that word occurs in the text, the method **isWordInUserDictionaries:caseSensitive:** will report YES rather than NO).
- Once the NSSpellServer delegate has reported a misspelled word, it has completed its search. Of course, it's likely that the user's application will then send a new message, this time asking the NSSpellServer to check a string containing the part of the text it didn't get to earlier.

Method Types

Registering your service

– registerLanguage:byVendor:

Assigning a delegate

– setDelegate:
– delegate

Running the service

– run

Checking user dictionaries

– isWordInUserDictionaries:caseSensitive:

Instance Methods

delegate

– (id)**delegate**

Returns the NSSpellServer's delegate.

See also: – **setDelegate:**

isWordInUserDictionaries:caseSensitive:

– (BOOL)**isWordInUserDictionaries:**(NSString *)*word* **caseSensitive:**(BOOL)*flag*

Indicates whether *word* is in the user's list of learned words or the document's list of words to ignore. If YES, the word is acceptable to the user. *flag* indicates whether the comparison is to be case-sensitive.

registerLanguage:byVendor:

– (BOOL)**registerLanguage:**(NSString *)*language* **byVendor:**(NSString *)*vendor*

Notifies the NSSpellServer of a language your spelling checker can check. *language* is the English name of a language on NeXT's list of languages. *vendor* identifies the vendor (to distinguish your spelling checker from those that others may offer for the same language). If your spelling checker supports more than one language, it should invoke this method once for each language. Registering a language/vendor combination causes it to appear in the Spelling Panel's pop-up list of spelling checkers.

Returns YES if the language is registered, NO if for some reason it can't be registered.

run

– (void)**run**

Causes the NSSpellServer to start listening for spell-checking requests. This method starts a loop that never returns; you need to set the NSSpellServer's delegate before sending this message.

See also: – **setDelegate:**

setDelegate:

– (void)**setDelegate:**(id)*anObject*

Assigns a delegate to the NSSpellServer. Since the delegate is where the real work is done, this is an essential step before telling the NSSpellServer to run.

See also: – **delegate**, – **run**

Methods Implemented by the Delegate

spellServer:didForgetWord:inLanguage:

– (void)**spellServer:**(NSSpellServer *)*sender*
didForgetWord:(NSString *)*word*
inLanguage:(NSString *)*language*

Notifies the delegate that *word* has been removed from the user's list of acceptable words. If your delegate maintains a similar auxiliary word list, you may wish to edit the list accordingly.

spellServer:didLearnWord:inLanguage:

– (void)**spellServer:**(NSSpellServer *)*sender*
didLearnWord:(NSString *)*word*
inLanguage:(NSString *)*language*

Notifies the delegate that *word* has been added to the user's list of acceptable words. If your delegate maintains a similar auxiliary word list, you may wish to edit the list accordingly.

spellServer:findMisspelledWordInString:language:wordCount:countOnly:

– (NSRange)**spellServer:**(NSSpellServer *)*sender*
findMisspelledWordInString:(NSString *)*stringToCheck*
language:(NSString *)*language*
wordCount:(int *)*wordCount*
countOnly:(BOOL)*countOnly*

Asks the delegate to search for a misspelled word in *stringToCheck*, using *language*, and marking the first misspelled word found by returning its range within the string object. In *wordCount*, return by reference the number of words from the beginning of the string object until the misspelled word (or the end-of-string). If *countOnly* is YES, just count the words in the string object; do not spell-check. Send **isWordInUserDictionaries:caseSensitive:** to the spelling server to determine if word exists in the user's language dictionaries.

spellServer:suggestGuessesForWord:inLanguage:

– (NSArray *)**spellServer:**(NSSpellServer *)*sender*
suggestGuessesForWord:(NSString *)*word*
inLanguage:(NSString *)*language*

Gives the delegate the opportunity to suggest guesses for the correct spelling of the misspelled *word*. Return the guesses as an array of NSStrings.

NSSplitView

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

Class Description

An NSSplitView object stacks several subviews within one view so that the user can change their relative sizes. By default, the split bars between the views are horizontal, so the views are one on top of the other. To have vertical split bars (so the views are side by side), use the method **setVertical:**. The rest of this section assumes you have horizontal split bars and gives information on vertical split bars in parentheses.

The NSSplitView resizes its subviews so that each subview is the same width (or height) as the NSSplitView, and the total of the subviews' heights (or widths), plus the total of the dividers' thicknesses, is equal to the height (or width) of the NSSplitView. The NSSplitView positions its subviews so that the first subview is at the top (or left) of the NSSplitView, and each successive subview is positioned below (or to the right of) the previous one. The user can set the height (or width) of two subviews by moving a horizontal (or vertical) bar called the *divider*, which makes one subview smaller and the other larger. Programmatically, you adjust the relative height of subviews simply by modifying the frame of each of the subviews.

Generally, you'll create an NSSplitView in Interface Builder by selecting the subviews and choosing the Group in Split View command. To add and remove subview to an NSSplitView programmatically, use the NSView methods **addSubview:** and **removeFromSuperview:**.

When a mouse-down occurs in an NSSplitView's divider, the NSSplitView determines the limits of the divider's travel and tracks the mouse to allow the user to drag the divider within these limits. With the following mouse-up, the NSSplitView resizes the two affected subviews, informs the delegate that the subviews were resized, and displays the affected views and divider. The NSSplitView's delegate can constrain the travel of specific dividers by implementing the method **splitView:constrainMinCoordinate: maxCoordinate:ofSubviewAt** or **splitView:constrainSplitPosition:ofSubviewAt:**

When the NSSplitView is displayed—whether it's being displayed for the first time or redisplayed because the user resized the NSSplitView's window—it checks to see if its subviews are properly tiled. If not, it invokes the delegate method **splitView:resizeSubviewsWithOldSize:**, allowing the delegate to specify the heights (or widths) of specific subviews. If the delegate doesn't implement this method, the NSSplitView sends **adjustSubviews** to itself to resize the subviews proportionately. Note that the NSSplitView doesn't call the delegate methods **splitView:constrainMinCoordinate:maxCoordinate:ofSubviewAt:** or

splitView:constrainSplitPosition:ofSubviewAt: and may resize its subviews in a way that isn't allowed by these delegate methods.

Method Types

Managing component views

– adjustSubviews
– dividerThickness
– drawDividerInRect:

Managing orientation

– isVertical
– setVertical:

Assigning a delegate

– delegate
– setDelegate:

Instance Methods

adjustSubviews

– (void)**adjustSubviews**

Adjusts the sizes of the NSSplitView's subviews so they (plus the dividers) fill the NSSplitView. The subviews are resized proportionally; the size of a subview relative to the other subviews doesn't change.

See also: – **setDelegate:**, – **setFrame:** (NSView)

delegate

– (id)**delegate**

Returns the NSSplitView's delegate.

dividerThickness

– (float)**dividerThickness**

Returns the thickness of the divider. The default thickness depends on the platform. You can subclass NSSplitView and override this method to change the divider's size, if necessary.

See also: – **drawDividerInRect:**

drawDividerInRect:

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

Draws the divider between two of the NSSplitView's subviews. *aRect* describes the entire divider rectangle in the NSSplitView's coordinates, which are flipped. The default implementation draws a default "dimple" image to the center of *aRect*; if you override this method and use a different icon to identify the divider, you may want to change the size of the divider.

See also: – **dividerThickness**, – **compositeToPoint:operation:** (NSImage)

isVertical

– (BOOL)**isVertical**

Returns YES if the split bars are vertical (subviews are side by side), NO if it they are horizontal (views are one on top of the other). By default, split bars are vertical.

See also: – **setVertical:**

setDelegate:

– (void)**setDelegate:**(id)*anObject*

Makes *anObject* the NSSplitView's delegate. The notification messages that the delegate can expect to receive are listed at the end of the NSSplitView class specification. The delegate doesn't need to implement all of the delegate methods.

setVertical:

– (void)**setVertical:**(BOOL)*flag*

Sets whether the split bars are vertical. If *flag* is YES, they're vertical (views are side by side); if it's NO, they're horizontal (views are one on top of the other).

See also: – **isVertical**

Methods Implemented By the Delegate

splitView:canCollapseSubview:

– (BOOL)**splitView:**(NSSplitView *)*sender* **canCollapseSubview:**(NSView *)*subview*

Allows the delegate to determine whether the user can collapse and un-collapse *subview*. If this method returns NO or is undefined, *subview* can't be collapsed. If this method returns YES, *subview* collapses when

the user drags a divider beyond the halfway mark between its minimum size and its edge. *subview* un-collapses when the user drags the divider back beyond that point. To specify the minimum size, define the method **splitView:constrainMinCoordinate:maxCoordinate:ofSubviewAt:**. Note that a subview can collapse only if you also define **splitView:constrainMinCoordinate:maxCoordinate:ofSubviewAt:**.

A collapsed subview is hidden but retained by the NSSplitView object, with the same size it had before it was collapsed

splitView:constrainMinCoordinate:maxCoordinate:ofSubviewAt:

– (void)**splitView:**(NSSplitView *)*sender* **constrainMinCoordinate:**(float *)*min* **maxCoordinate:**(float *)*max* **ofSubviewAt:**(int)*offset*

Allows the delegate to constrain the coordinate limits of a divider when the user drags it. Invoke this method before the NSSplitView begins tracking the mouse to position a divider. You may further constrain the limits that have been already set, but you cannot extend the divider limits. *min* and *max* are specified in the NSSplitView's flipped coordinate system. If the split bars are horizontal (views are one on top of the other), *min* is the top limit and *max* is the bottom limit. If the split bars are vertical (views are side by side), *min* is the left limit and *max* is the right limit. The initial value of *min* is top (or left side) of the subview before the divider, and the initial value of *max* is the bottom (or right side) of the subview after the divider. *offset* specifies the divider the user is moving, with the first divider being 0 and going from top to bottom (or from left to right).

See also: – **isVertical**

splitView:constrainSplitPosition:ofSubviewAt:

– (float)**splitView**:(NSSplitView *)*splitView*
constrainSplitPosition:(float)*proposedPosition*
ofSubviewAt:(int)*offset*

Allows the delegate to constrain the divider to certain positions. If the delegate implemements this method, the NSSplitView calls it repeatedly as the user moves the divider. This method returns where you want the divider to be, given *proposedPosition*, the mouse's current position. *offset* is the divider the user is moving, with the first divider being 0 and going from top to bottom (or from left to right).

For example, if a subview's height must be a multiple of a certain number, use this method to return the multiple nearest to *proposedPosition*.

splitView:resizeSubviewsWithOldSize:

– (void)**splitView:**(NSSplitView *)*sender* **resizeSubviewsWithOldSize:**(NSSize)*oldSize*

Allows the delegate to specify custom sizing behavior for the subviews of the NSSplitView. If the delegate implements this method, **splitView:resizeSubviewsWithOldSize:** is invoked after the NSSplitView is resized. The size of the NSSplitView before the user resized it is indicated by *oldSize*; the subviews should be resized such that the sum of the sizes of the subviews plus the sum of the thickness of the dividers equals the size of the NSSplitView's new frame. You can get the thickness of a divider through the **dividerThickness** method.

Note that if you implement this delegate method to resize subviews on your own, the NSSplitView does not perform any error checking for you. However, you can invoke **adjustSubviews** to perform the default sizing behavior.

See also: – **adjustSubviews**, – **setFrame:** (NSView)

splitViewDidResizeSubviews:

– (void)**splitViewDidResizeSubviews:**(NSNotification *)*aNotification*

Sent by the default notification center to the delegate; *aNotification* is always an NSSplitViewDidResizeSubviewsNotification. If the delegate implements this method, the delegate is automatically registered to receive this notification. This method is invoked after the NSSplitView resizes two of its subviews in response to the repositioning of a divider.

splitViewWillResizeSubviews:

– (void)**splitViewWillResizeSubviews:**(NSNotification *)*aNotification*

Sent by the default notification center to the delegate; *aNotification* is always an NSSplitViewWillResizeSubviewsNotification. If the delegate implements this method, the delegate is automatically registered to receive this notification. This method is invoked before the NSSplitView resizes two of its subviews in response to the repositioning of a divider.

Notifications

NSSplitView declares and posts the following notifications. In addition, it posts notifications that are declared by its superclass, NSView. See the NSView class specification for more information.

NSSplitViewDidResizeSubviewsNotification

This notification contains a notification object but no userInfo dictionary. The notification object is the NSSplitView that resized its subviews.

Posted after the NSSplitView changes the sizes of some or all of its subviews.

See also: **– splitViewDidResizeSubviews:**

NSSplitViewWillResizeSubviewsNotification

This notification contains a notification object but no userInfo dictionary. The notification object is the NSSplitView object that is about to resize its subviews.

Posted before the NSSplitView changes the sizes of some or all of its subviews.

See also: **– splitViewWillResizeSubviews:**

NSString Additions

Inherits From: NSObject

Declared In: AppKit/NSStringDrawing.h

Class Description

The Application Kit adds three methods to the NSString class to support drawing string objects directly in an NSView: **drawAtPoint:withAttributes:**, **drawInRect:withAttributes:**, and **sizeWithAttributes:**. The Application Kit adds similar method the NSAttributedString class. The two drawing methods draw a string object with a single set of attributes that apply to the entire string. To draw a string with multiple attributes, such as multiple text fonts, you must use an NSAttributedString.

Method Types

Drawing an NSString

– drawAtPoint:withAttributes:
– drawInRect:withAttributes:
– sizeWithAttributes:

Instance Methods

drawAtPoint:withAttributes:

– (void)**drawAtPoint:**(NSPoint)*aPoint* **withAttributes:**(NSDictionary *)*attributes*

Draws the receiver with the font and other display characteristics of *attributes*, at *aPoint* in the currently focused NSView. You should only invoke this method when an NSView has PostScript focus.

See also: – **lockFocus** (NSView)

drawInRect:withAttributes:

– (void)**drawInRect:**(NSRect)*aRect* **withAttributes:**(NSDictionary *)*attributes*

Draws the receiver with the font and other display characteristics of *attributes*, within *aRect* in the currently focused NSView. You should only invoke this method when an NSView has PostScript focus.

See also: – **lockFocus** (NSView)

sizeWithAttributes:

– (NSSize)**sizeWithAttributes:**(NSDictionary *)*attributes*

Returns the bounding box size that the receiver occupies when drawn with *attributes*.

NSTableColumn

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: AppKit/NSTableColumn.h

Class Description

An NSTableColumn stores the display characteristics and attribute identifier for a column in an NSTableView. The NSTableColumn determines the width and width limits, resizability, and editability of its column in the NSTableView. It also stores two NSCell objects: the *header cell*, which is used to draw the column header, and the *data cell*, used to draw the values for each row. You can control the display of the column by setting the subclasses of NSCell used and by setting the font and other display characteristics for these NSCells. For example, you can use the default NSTextFieldCell for displaying string values or substitute an NSImageCell to display pictures.

See the NSTableView class specification for a general overview.

Method Types

Creating an NSTableColumn instance

– initWithIdentifier:

Setting the identifier

– setIdentifier:
– identifier

Setting the NSTableView

– setTableView:
– tableView

Controlling size

– setWidth:
– width
– setMinWidth:
– minWidth
– setMaxWidth:
– maxWidth
– setResizable:
– isResizable
– sizeToFit

Controlling editability

– setEditable:
– isEditable

Setting component cells

– setHeaderCell:
– headerCell
– setDataCell:
– dataCell

Instance Methods

dataCell

– (id)**dataCell**

Returns the NSCell object used by the NSTableView to draw values for the NSTableColumn.

See also: – **setDataCell:**

headerCell

– (id)**headerCell**

Returns the NSTableHeaderCell object used to draw the header of the NSTableColumn. You can set the column title by sending **setStringValue:** to this object.

See also: – **setHeaderCell:**

initWithIdentifier:

– (id)**initWithIdentifier:***anObject*

Initializes a newly created NSTableColumn with *anObject* as its identifier and with an NSTextFieldCell as its data cell. Send **setStringValue:** to the header cell to set the column title. This is the designated initializer for the NSTableColumn class. Returns **self**.

See the NSTableView class specification for information on identifiers.

See also: – **setIdentifier:**

identifier

– (id)**identifier**

Returns the object used by the data source to identify the attribute corresponding to the NSTableColumn.

See also: – **setIdentifier:**

isEditable

– (BOOL)**isEditable**

Returns YES if the user can edit cells associated with the NSTableColumn by double-clicking the column in the NSTableView, NO otherwise. You can initiate editing programmatically regardless of this setting with NSTableView's **editColumn:row:withEvent:select:** method.

See also: – **setEditable:**

isResizable

– (BOOL)**isResizable**

Returns YES if the user is allowed to resize the NSTableColumn in its NSTableView, NO otherwise. You can change the size programmatically regardless of this setting.

See also: – **setWidth:**, – **setMinWidth:**, – **setMaxWidth:**, – **setResizable:**

maxWidth

– (float)**maxWidth**

Returns the maximum width for the NSTableColumn. The NSTableColumn's width can't be made larger than this either by the user or programmatically.

See also: – **minWidth**, – **width**, – **setMaxWidth:**, – **sizeToFit** (NSTableView), – **autoresizesAllColumnsToFit** (NSTableView)

minWidth

– (float)**minWidth**

Returns the minimum width for the NSTableColumn. The NSTableColumn's width can't be made less than this either by the user or programmatically.

See also: – **maxWidth**, – **width**, – **setMinWidth:**, – **sizeToFit** (NSTableView), – **autoresizesAllColumnsToFit** (NSTableView)

setDataCell:

– (void)**setDataCell:**(NSCell *)*aCell*

Sets the NSCell used by the NSTableView to draw individual values for the NSTableColumn to *aCell*. You can use this method to control the font, alignment, and other text attributes for an NSTableColumn. You can also assign a cell to display things other than text—for example, an NSImageCell to display images.

See also: – **dataCell**

setEditable:

– (void)**setEditable:**(BOOL)*flag*

Controls whether the user can edit cells in the receiver by double-clicking them. If *flag* is YES a double click initiates editing; if *flag* is NO it merely sends the double action to the NSTableView's target. You can initiate editing programmatically regardless of this setting with NSTableView's **editColumn:row:withEvent: select:** method.

See also: – **isEditable**

setHeaderCell:

– (void)**setHeaderCell:**(NSCell *)*aCell*

Sets the NSCell used to draw the NSTableColumn's header to *aCell*. *aCell* should never be **nil**.

See also: – **headerCell**

setIdentifier:

– (void)**setIdentifier:**(id)*anObject*

Sets the NSTableColumn's identifier to *anObject*. This object is used by the data source to identify the attribute corresponding to the NSTableColumn.

See also: – **identifier**

setMaxWidth:

– (void)**setMaxWidth:**(float)*maxWidth*

Sets the NSTableColumn's maximum width to *maxWidth*, also adjusting the current width if it's greater than this value. The NSTableView can be made no wider than this, either by the user or programmatically.

See also: – **setMinWidth:**, – **setWidth:**, – **maxWidth**, – **sizeToFit** (NSTableView),
– **autoresizesAllColumnsToFit** (NSTableView)

setMinWidth:

– (void)**setMinWidth:**(float)*minWidth*

Sets the NSTableColumn's minimum width to *minWidth*, also adjusting the current width if it's less than this value. The NSTableView can be made no less wide than this, either by the user or programmatically.

See also: – **setMaxWidth:**, – **setWidth:**, – **minWidth**, – **sizeToFit** (NSTableView),
– **autoresizesAllColumnsToFit** (NSTableView)

setResizable:

– (void)**setResizable:**(BOOL)*flag*

Sets whether the user can resize the receiver in its NSTableView. If *flag* is YES the user can resize the receiver; if *flag* is NO the user can't resize it (though you can set the size programmatically).

See also: – **isResizable**, – **setWidth:**, – **setMinWidth:**, – **setMaxWidth:**

setTableView:

– (void)**setTableView:**(NSTableView *)*aTableView*

Sets *aTableView* as the NSTableColumn's NSTableView. You should never need to invoke this method; it's invoked automatically when you add an NSTableColumn to an NSTableView.

See also: – **tableView**, – **addTableColumn:** (NSTableView)

setWidth:

– (void)**setWidth:**(float)*newWidth*

Sets the NSTableColumn's width to *newWidth*. If *newWidth* exceeds the minimum or maximum width, it's adjusted to the appropriate limiting value. Marks the NSTableView as needing display.

This method posts NSTableViewColumnDidResizeNotification on behalf of the NSTableColumn's NSTableView.

See also: – **width**, – **setMinWidth:**, – **setMaxWidth:**, – **sizeToFit** (NSTableView),
– **autoresizesAllColumnsToFit** (NSTableView)

sizeToFit

– (void)**sizeToFit**

Resizes the NSTableColumn to fit the width of its header cell. If the maximum width is less than the width of the header, the maximum is increased to the header's width. Similarly, if the minimum width is greater than the width of the header, the minimum is reduced to the header's width. Marks the NSTableView as needing display if the width actually changes.

See also: – **width**, – **minWidth**, – **maxWidth**, – **sizeToFit** (NSTableView),
– **autoresizesAllColumnsToFit** (NSTableView)

tableView

– (NSTableView *)**tableView**

Returns the NSTableView that the NSTableColumn belongs to.

See also: – **setTableView:**

width

– (float)**width**

Returns the width of the NSTableColumn.

See also: – **width**

NSTableHeaderCell

Inherits From: NSTextFieldCell : NSActionCell : NSCell : NSObject

Conforms To: NSCoding (NSCell)
NSCopying (NSCell)
NSObject (NSObject)

Declared In: AppKit/NSTableHeaderCell.h

Class Description

An NSTableHeaderCell is used by an NSTableHeaderView to draw its column headers. See the NSTableView class specification for more information on how it's used.

Subclasses of NSTableHeaderCell can override **drawInteriorWithFrame:inView:**, **drawWithFrame: inView:**, and **highlight:withFrame:inView:** to change the way headers appear. See the NSCell class specification, and the description below, for information on these methods.

Instance Methods

drawInteriorWithFrame:inView:

– (void)**drawInteriorWithFrame:**(NSRect)*cellFrame* **inView:**(NSView *)*controlView*

Draws the receiver's interior, as described for this same method in the NSCell class specification. NSTableHeaderCell's implementation overrides NSTextFieldCell's to draw the receiver's image if it has one, instead of its string value. If the receiver has no image, it simply draws its string value. This allows column headers to be labeled with images rather than text.

To make an NSTableHeaderCell display an image, use NSCell's **setImage:** method, which changes the receiver's cell type to NSImageCellType and stores the image provided. To restore it to displaying a text label, supply a new title using **setStringValue:**, which removes the image and reverts the receiver's cell type to NSTextCellType.

NSTableHeaderView

Inherits From: NSView : NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSTableHeaderView.h

Class Description

An NSTableHeaderView is used by an NSTableView to draw headers over its columns and to handle mouse events in those headers. See the NSTableView class specification for more information.

Method Types

Setting the table view

– setTableView:
– tableView

Checking altered columns

– draggedColumn
– draggedDistance
– resizedColumn

Utility methods

– columnAtPoint:
– headerRectOfColumn:

Instance Methods

columnAtPoint:

– (int)**columnAtPoint:**(NSPoint)*aPoint*

Returns the index of the column whose header lies under *aPoint* in the NSTableHeaderView, or –1 if no such column is found. *aPoint* is expressed in the NSTableHeaderView's coordinate system.

draggedColumn

– (int)**draggedColumn**

If the user is dragging a column in the NSTableView, returns the index of that column. Otherwise returns –1.

See also: – **draggedDistance**

draggedDistance

– (float)**draggedDistance**

If the user is dragging a column in the NSTableView, returns the column's horizontal distance from its original position. Otherwise the return value is meaningless.

See also: – **draggedColumn**

headerRectOfColumn:

– (NSRect)**headerRectOfColumn:**(int)*columnIndex*

Returns the rectangle containing the header tile for the column at *columnIndex*. Raises an NSInternalInconistencyException if *columnIndex* is out of bounds.

See also: – **rectOfColumn:** (NSTableView)

resizedColumn

– (int)**resizedColumn**

If the user is resizing a column in the NSTableView, returns the index of that column. Otherwise returns –1.

setTableView:

– (void)**setTableView:**(NSTableView *)*aTableView*

Sets *aTableView* as the NSTableColumn's NSTableView. You should never need to invoke this method; it's invoked automatically when you set the header view for an NSTableView.

See also: – **setHeaderView:** (NSTableView)

tableView

– (NSTableView *)**tableView**

Returns the NSTableView that the NSTableHeaderView belongs to.

NSTableView

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

Class at a Glance

Purpose

An NSTableView object displays record-oriented data in a table, and allows the user to edit values and resize and rearrange columns.

Principal Attributes

Displays record-oriented data	Works with NSScrollView
Gets data from an object you provide	Uses a delegate
Lazily retrieves only data that needs to be displayed	

Creation

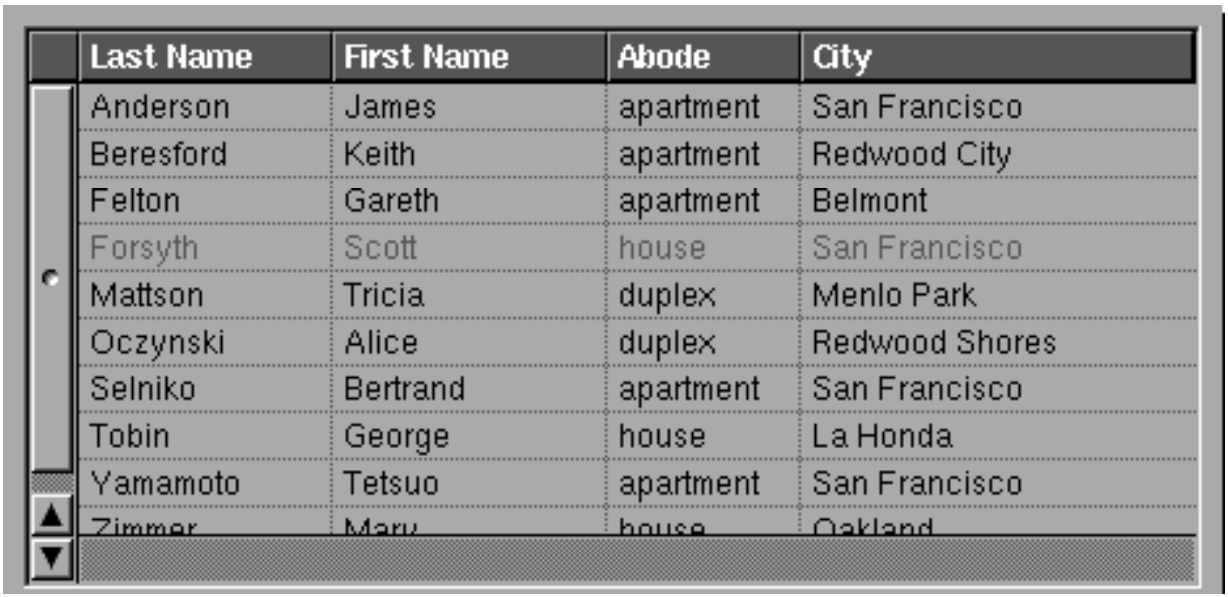
Interface Builder	
– initWithFrame:	Designated initializer

Commonly Used Methods

– dataSource	Returns the object providing the data that the NSTableView displays.
– tableColumns	Returns the NSTableColumn objects representing attributes for the NSTableView.
– selectedColumn	Returns the index of the selected column.
– selectedRow	Returns the index of the selected column.
– numberOfRows	Returns the number of rows in the NSTableView.
– reloadData	Informs the NSTableView that data has changed and needs to be retrieved and displayed again.

Class Description

An NSTableView displays data for a set of related records, with rows representing individual records and columns representing the attributes of those records. A record is a set of values for a particular real-world entity, such as an employee or a bank account. For example, in a table of employee records, each row represents one employee, and the columns represent such attributes as the first and last name, address, salary, and so on. An NSTableView is usually displayed in an NSScrollView, like this:



Last Name	First Name	Abode	City
Anderson	James	apartment	San Francisco
Beresford	Keith	apartment	Redwood City
Felton	Gareth	apartment	Belmont
Forsyth	Scott	house	San Francisco
Mattson	Tricia	duplex	Menlo Park
Oczynski	Alice	duplex	Redwood Shores
Selniko	Bertrand	apartment	San Francisco
Tobin	George	house	La Honda
Yamamoto	Tetsuo	apartment	San Francisco
Zimmer	Mary	house	Oakland

In this illustration, the NSTableView itself is only the portion displaying values. The header is drawn by two auxiliary views: the column headers by the *header view*, and the blank square above the vertical scroller by the *corner view*. The roles of these two auxiliary views are discussed in ““Auxiliary Components”.”

The user selects rows or columns in the table by clicking, and edits individual cells by double-clicking. The user can also rearrange columns by dragging the column headers and can resize the columns by dragging the divider between two column headers. You can configure the table’s parameters so that the user can select more than one row or column (or have none selected), so that the user isn’t allowed to edit particular columns or rearrange them, and so on. You can also specify an action message to be sent when the user double-clicks something other than an editable cell.

Providing Data for Display

Unlike most NSControls, an NSTableView doesn’t store or cache the data it displays. Instead, it gets all of its data from an object that you provide, called its *data source*. Your data source object can store records in any way, but it must be able to identify them by integer index and must implement methods to provide the following information: how many records the data source contains, and what the value is for a particular record’s attribute. If you want to allow the user to edit the records, you must also provide a method for changing the value of an attribute. These methods are described in the NSTableDataSource informal protocol specification.

A record attribute is indicated by an object called its *identifier*, which is associated with a column in the NSTableView, as described in ““Auxiliary Components”.” Because a column can be reordered, its index can’t be used to identify a record attribute. Instead, the data source uses the column’s identifier as a key to retrieve the value for a column’s attribute. The identifier can be any kind of object that uniquely identifies attributes for the data source. For example, if you specify identifiers as NSStrings containing the names of

attributes, such as “Last Name”, “Address”, and so on, the data source object can use these strings as keys into NSDictionary objects. See the NSTableDataSource informal protocol specification for an example of how to use identifiers.

Auxiliary Components

As indicated earlier, an NSTableView is usually displayed in an NSScrollView along with its two auxiliary views, the corner view and the header view. The corner view is by default a simple view that merely fills in the corner above the vertical scroller. You can replace the default corner view with a custom view; for example, a button that sorts based on the selected column. The header view is usually an instance of the NSTableHeaderView class, which draws the column headers and handles column selection, rearranging, and resizing. NSScrollView queries any document view it’s given for the **cornerView** and **headerView** methods, and if the document view responds and returns objects for them, the NSScrollView automatically tiles them along with its scrollers and the document view.

The NSTableView and the NSTableHeaderView both need access to information about columns (such as their width), so this information is encapsulated in NSTableColumn objects. An NSTableColumn stores its column’s width, and determines whether the user can resize the column or edit its cells. It also holds an NSCell object that the NSTableHeaderView uses to draw the column header, and an NSCell object that the NSTableView uses to draw values in the column (it reuses the same NSCell for each row in the column). Finally, the NSTableColumn holds the attribute identifier mentioned in ““Providing Data for Display”.”

The cell for each column header is by default an instance of the NSTableHeaderCell class; it’s used by the NSTableHeaderView to draw the column’s header. An NSTableHeaderCell contains the title displayed over the column, as well as the font and color for that title. You use the API of its superclasses, NSTextFieldCell and NSCell, to set a column’s title and to specify display attributes for that title (font, alignment, and so on). In addition, you can use the NSCell method **setImage:** to make the NSTableHeaderCell display an image instead of a title. To remove the image and restore the title, use the NSCell method **setStringValue:**.

The data cell for the column values is typically an instance of NSTextFieldCell, but can be an instance of any NSCell subclass, such as NSImageCell. This object is used to draw all values in the column and determines the font, alignment, text color, and other such display attributes for those values. You can customize the presentation of various kinds of values by assigning an NSFormatter to the cell. For example, to properly display NSDate values in a column, assign its data cell an NSDateFormatter.

Delegate Messages

NSTableView adds a handful of delegate messages to those defined by its superclass, NSControl. These methods give the delegate control over the appearance of individual cells in the table, over changes in selection, and over editing of cells. Delegate methods that request permission to alter the selection or edit a value are invoked during user actions that affect the NSTableView, but are not invoked by programmatic changes to the view. When making changes programmatically, you decide whether you want the delegate to intervene and, if so, send the appropriate message (checking first that the delegate responds to that

message). Because the delegate methods involve the actual data displayed by the NSTableView, the delegate is typically the same object as the data source.

tableView:willDisplayCell:forTableColumn:row: informs the delegate that the NSTableView is about to draw a particular cell. The delegate can modify the NSCell provided to alter the display attributes for that cell; for example, making uneditable values display in italic or gray text (as in the figure above).

tableView:shouldSelectRow: and **tableView:shouldSelectTableColumn:** give the delegate control over whether the user can select a particular row or column (though the user can still reorder columns). This is useful for disabling particular rows or columns. For example, in a database client application, when another user is editing a record you might want all other users not to be able to select it.

selectionShouldChangeInTableView: allows the delegate to deny a change in selection; for example, if the user is editing a cell and enters an improper value, the delegate can prevent the user from selecting or editing any other cells until a proper value has been entered into the original cell.

tableView:shouldEditTableColumn:row: asks the delegate whether it's okay to edit a particular cell. The delegate can approve or deny the request.

In addition to these methods, the delegate is also automatically registered to receive messages corresponding to NSTableView notifications. These inform the delegate when the selection changes and when a column is moved or resized:

Delegate Message	Notification
tableViewColumnDidMove:	NSTableViewColumnDidMoveNotification
tableViewColumnDidResize:	NSTableViewColumnDidResizeNotification
tableViewSelectionDidChange:	NSTableViewSelectionDidChangeNotification
tableViewSelectionIsChanging:	NSTableViewSelectionIsChangingNotification

Method Types

Creating an instance

– initWithFrame:

Setting the data source

– setDataSource:
– dataSource

Loading data

– reloadData

Target-action behavior

– setDoubleAction:
– doubleAction
– clickedColumn
– clickedRow

Configuring behavior

– setAllowsColumnReordering:
– allowsColumnReordering
– setAllowsColumnResizing:
– allowsColumnResizing
– setAllowsMultipleSelection:
– allowsMultipleSelection
– setAllowsEmptySelection:
– allowsEmptySelection
– setAllowsColumnSelection:
– allowsColumnSelection

Setting display attributes

– setIntercellSpacing:
– intercellSpacing
– setRowHeight:
– rowHeight
– setBackgroundColor:
– backgroundColor

Manipulating columns

– addTableColumn:
– removeTableColumn:
– moveColumn:toColumn:
– tableColumns
– columnWithIdentifier:
– tableColumnWithIdentifier:

Selecting columns and rows

– selectColumn:byExtendingSelection:
– selectRow:byExtendingSelection:
– deselectColumn:
– deselectRow:
– numberOfSelectedColumns
– numberOfSelectedRows
– selectedColumn
– selectedRow
– isColumnSelected:
– isRowSelected:
– selectedColumnEnumerator
– selectedRowEnumerator
– selectAll:
– deselectAll:

Getting the dimensions of the table

– numberOfColumns
– numberOfRows

Setting grid attributes

– setDrawsGrid:
– drawsGrid
– setGridColor:
– gridColor

Editing cells

– editColumn:row:withEvent:select:
– editedRow
– editedColumn

Setting auxiliary views

– setHeaderView:
– headerView
– setCornerView:
– cornerView

Layout support

– rectOfColumn:
– rectOfRow:
– columnsInRect:
– rowsInRect:
– columnAtPoint:
– rowAtPoint:
– frameOfCellAtColumn:row:
– setAutoresizesAllColumnsToFit:
– autoresizesAllColumnsToFit
– sizeLastColumnToFit
– sizeToFit
– noteNumberOfRowsChanged
– tile

Drawing

– drawRow:clipRect:
– drawGridInClipRect:
– highlightSelectionInClipRect:

Scrolling

– scrollRowToVisible:
– scrollColumnToVisible:

Text delegate methods

– textShouldBeginEditing:
– textDidBeginEditing:
– textDidChange:
– textShouldEndEditing:
– textDidEndEditing:

Persistence

– autosaveName
– autosaveTableColumns
– setAutosaveName:
– setAutosaveTableColumns:

Setting the delegate

– setDelegate:
– delegate

Instance Methods

addTableColumn:

– (void)**addTableColumn:**(NSTableColumn *)*aColumn*

Appends *aColumn* to the receiver.

See also: – **sizeLastColumnToFit**, – **sizeToFit**, – **removeTableColumn:**

allowsColumnReordering

– (BOOL)**allowsColumnReordering**

Returns YES if the receiver allows the user to rearrange columns by dragging their headers, NO otherwise. The default is YES. You can rearrange columns programmatically regardless of this setting.

See also: – **moveColumn:toColumn:**, – **setAllowsColumnReordering:**

allowsColumnResizing

– (BOOL)**allowsColumnResizing**

Returns YES if the receiver allows the user to resize columns by dragging between their headers, NO otherwise. The default is YES. You can resize columns programmatically regardless of this setting.

See also: – **setWidth:** (NSTableColumn), – **setAllowsColumnResizing:**

allowsColumnSelection

– (BOOL)**allowsColumnSelection**

Returns YES if the receiver allows the user to select columns by clicking their headers, NO otherwise. The default is YES. You can select columns programmatically regardless of this setting.

See also: – **selectColumn:byExtendingSelection:**, – **allowsColumnReordering**,
– **setAllowsColumnSelection:**

allowsEmptySelection

– (BOOL)**allowsEmptySelection**

Returns YES if the receiver allows the user to select zero columns or rows, NO otherwise. The default is YES.

You can *not* set an empty selection programmatically if this setting is NO, unlike with the other settings that affect selection behavior.

See also: – **deselectAll:**, – **deselectColumn:**, – **deselectRow:**, – **setAllowsEmptySelection:**

allowsMultipleSelection

– (BOOL)**allowsMultipleSelection**

Returns YES if the receiver allows the user to select more than one column or row at a time, NO otherwise. The default is NO. You can select multiple columns or rows programmatically regardless of this setting.

See also: – **selectColumn:byExtendingSelection:**, – **selectRow:byExtendingSelection:**, – **setAllowsMultipleSelection:**

autoresizesAllColumnsToFit

– (BOOL)**autoresizesAllColumnsToFit**

Returns YES if the receiver proportionally resizes its columns to fit when its superview's frame changes, NO if it only resizes the last column.

See also: – **setAutoresizesAllColumnsToFit:**, – **sizeLastColumnToFit**, – **sizeToFit**

autosaveName

– (NSString *)**autosaveName**

Returns the name under which table information is automatically saved. If no name has been set, this method returns **nil**. The table information is saved separately for each user and for each application that user uses.

Note that even when a table view has an autosave name, it may not be saving table information automatically . To check whether table information is being saved automatically, use **autosaveTableColumns**.

See also: – **autosaveTableColumns**, – **setAutosaveName:**

autosaveTableColumns

– (BOOL)**autosaveTableColumns**

Returns whether the order and width of this table view's columns are automatically saved.

The table information is saved separately for each user and for each application that user uses. Note that if **autosaveName** returns **nil**, this setting is ignored and table information isn't saved.

See also: – **autosaveName**, – **setAutosaveTableColumns:**, – **setAutosaveName:**

backgroundColor

– (NSColor *)**backgroundColor**

Returns the color used to draw the background of the receiver. The default background color is light gray.

See also: – **setBackgroundColor:**

clickedColumn

– (int)**clickedColumn**

Returns the index of the column the user clicked to trigger an action message. The return value of this method is meaningful only in the target's implementation of the action or double-action method.

See also: – **clickedRow**, – **setAction:** (NSControl), – **setDoubleAction:**

clickedRow

– (int)**clickedRow**

Returns the index of the row the user clicked to trigger an action message. The return value of this method is meaningful only in the target's implementation of the action or double-action method.

See also: – **clickedColumn**, – **setAction:** (NSControl), – **setDoubleAction:**

columnAtPoint:

– (int)**columnAtPoint:**(NSPoint)*aPoint*

Returns the index of the column that *aPoint* lies in, or –1 if *aPoint* lies outside the receiver's bounds.

See also: – **rowAtPoint:**

columnsInRect:

– (NSRange)**columnsInRect:**(NSRect)*aRect*

Returns a range of indices for the receiver's columns that lie wholly or partially within the horizontal boundaries of *aRect*; the location of the range is the first such column's index, and the length is the number

of columns that lie in *aRect*. Both the width and height of *aRect* must be nonzero values, or **columnsInRect:** returns an NSRange whose length is zero.

See also: – **rowsInRect:**

columnWithIdentifier:

– (int)**columnWithIdentifier:**(id)*anObject*

Returns the index of the first column in the receiver whose identifier is equal to *anObject*, when compared using **isEqual:**, or –1 if no columns are found with the specified identifier.

See also: – **tableColumnWithIdentifier:**

cornerView

– (NSView *)**cornerView**

Returns the NSView used to draw the area to the left of the column headers and above the vertical scroller of the enclosing NSScrollView. This is by default a simple view that merely fills in its frame, but you can replace it with a custom view using **setCornerView:**.

See also: – **headerView**

dataSource

– (id)**dataSource**

Returns the object that provides the data displayed by the receiver. See the class description and the NSTableDataSource informal protocol specification for more information.

See also: – **setDataSource:**

delegate

– (id)**delegate**

Returns the receiver's delegate.

See also: – **setDelegate:**

deselectAll:

– (void)**deselectAll:**(id)*sender*

Deselects all selected rows or columns if empty selection is allowed, otherwise does nothing. Posts NSTableViewSelectionDidChangeNotification to the default notification center if the selection does in fact change.

As a target-action method, **deselectAll:** checks with the delegate before changing the selection, using **selectionShouldChangeInTableView:**.

See also: – **allowsEmptySelection**, – **selectAll:**, – **selectColumn:byExtendingSelection:**

deselectColumn:

– (void)**deselectColumn:**(int)*columnIndex*

Deselects the column at *columnIndex* if it's selected, regardless of whether empty selection is allowed. If the selection does in fact change, posts NSTableViewSelectionDidChangeNotification to the default notification center.

If the indicated column was the last column selected by the user, the column nearest it effectively becomes the last selected column. In case of a tie, priority is given to the column on the left.

This method doesn't check with the delegate before changing the selection.

See also: – **selectedColumn**, – **allowsEmptySelection**, – **selectRow:byExtendingSelection:**

deselectRow:

– (void)**deselectRow:**(int)*rowIndex*

Deselects the row at *rowIndex* if it's selected, regardless of whether empty selection is allowed. If the selection does in fact change, posts NSTableViewSelectionDidChangeNotification to the default notification center.

If the indicated row was the last row selected by the user, the row nearest it effectively becomes the last selected row. In case of a tie, priority is given to the row above.

This method doesn't check with the delegate before changing the selection.

See also: – **selectedRow**, – **allowsEmptySelection**

doubleAction

– (SEL)**doubleAction**

Returns the message sent to the target when the user double-clicks a column header or an uneditable cell.

See also: – **action** (NSControl), – **target** (NSControl), – **setDoubleAction:**

drawGridInClipRect:

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

Draws the grid lines within *aRect*, using the grid color set with **setGridColor:**. This method draws a grid regardless of whether the receiver is set to draw one automatically.

Subclasses can override this method to draw grid lines other than the standard ones.

See also: – **gridColor**, – **setIntercellSpacing:**, – **drawsGrid**, – **drawRow:clipRect:**, – **highlightSelectionInClipRect:**

drawRow:clipRect:

– (void)**drawRow:**(int)*rowIndex* **clipRect:**(NSRect)*clipRect*

Draws the cells for the row at *rowIndex* in the columns that intersect *clipRect*. Sends **tableView: willDisplayCell:forTableColum:row:** to the delegate before drawing each cell.

Subclasses can override this method to customize their appearance.

See also: – **columnsInRect:**, – **highlightSelectionInClipRect:**, – **drawGridInClipRect:**

drawsGrid

– (BOOL)**drawsGrid**

Returns YES if the receiver draws grid lines around cells, NO if it doesn't. The default is YES.

See also: – **gridColor**, – **drawGridInClipRect:**, – **setDrawsGrid:**

editColumn:row:withEvent:select:

– (void)**editColumn:**(int)*columnIndex*
row:(int)*rowIndex*
withEvent:(NSEvent *)*theEvent*
select:(BOOL)*flag*

Edits the cell at *columnIndex* and *rowIndex*, selecting its entire contents if *flag* is YES. This method is invoked automatically in response to user actions; you should rarely need to invoke it directly. *theEvent* is usually the mouse event that triggered editing; it can be **nil** when starting an edit programmatically.

This method scrolls the receiver so that the cell is visible, sets up the field editor, and sends **selectWithFrame:inView:editor:delegate:start:length:** and **editWithFrame:inView:editor:delegate:event:** to the field editor's NSCell object with the NSTableView as the text delegate.

See also: – **editedColumn**, – **editedRow**

editedColumn

– (int)**editedColumn**

If sent during **editColumn:row:withEvent:select:** returns the index of the column being edited; otherwise returns –1.

editedRow

– (int)**editedRow**

If sent during **editColumn:row:withEvent:select:** returns the index of the row being edited; otherwise returns –1.

frameOfCellAtColumn:row:

– (NSRect)**frameOfCellAtColumn:**(int)*columnIndex* **row:**(int)*rowIndex*

Returns a rectangle locating the cell that lies at the intersection of *columnIndex* and *rowIndex*. Returns NSZeroRect if *columnIndex* or *rowIndex* are greater than the number of columns or rows in the NSTableView.

The result of this method is used in a **drawWithFrame:inView:** message to the NSTableColumn's data cell.

See also: – **rectOfColumn:**, – **rectOfRow:**

gridColor

– (NSColor *)**gridColor**

Returns the color used to draw grid lines. The default color is gray.

See also: – **drawsGrid**, – **drawGridInClipRect:**, – **setGridColor:**

headerView

– (NSTableHeaderView *)**headerView**

Returns the NSTableHeaderView used to draw headers over columns, or **nil** if the NSTableView has no header view. See the class description and the NSTableHeaderView class specification for more information.

See also: – **setHeaderView:**

highlightSelectionInClipRect:

– (void)**highlightSelectionInClipRect:**(NSRect)*clipRect*

Highlights the region of the receiver in *clipRect*. This method is invoked before **drawRow:clipRect:**.

Subclasses can override this method to change the manner in which they highlight selections.

See also: – **drawGridInClipRect:**

initWithFrame:

– (id)**initWithFrame:**(NSRect)*frameRect*

Initializes a newly allocated NSTableView with *frameRect* as its frame rectangle. The new NSTableView has a header view but has no columns; you can create NSTableColumn objects, set their titles and attributes, and add them to the new NSTableView with **addTableColumn:**. You must also set the NSTableView up in an NSScrollView with NSScrollView's **setDocView:** method. This is the designated initializer for the NSTableView class. Returns **self**.

It's usually more convenient to create an NSTableView using Interface Builder. Interface Builder lets you create an NSTableView already embedded in an NSScrollView, add and name the columns, and set up a data source.

intercellSpacing

– (NSSize)**intercellSpacing**

Returns the horizontal and vertical spacing between cells. The default spacing is (3.0, 2.0).

See also: – **setDrawsGrid:**, **– setIntercellSpacing:**

isColumnSelected:

– (BOOL)**isColumnSelected:**(int)*columnIndex*

Returns YES if the column at *columnIndex* is selected, NO otherwise.

See also: – **selectedColumn**, – **selectedColumnEnumerator**, – **selectColumn:byExtendingSelection:**

isRowSelected:

– (BOOL)**isRowSelected:**(int)*rowIndex*

Returns YES if the row at *rowIndex* is selected, NO otherwise.

See also: – **selectedRow**, – **selectedRowEnumerator**, – **selectRow:byExtendingSelection:**

moveColumn:toColumn:

– (void)**moveColumn:**(int)*columnIndex* **toColumn:**(int)*newIndex*

Moves the column and heading at *columnIndex* to *newIndex*.

This method posts NSTableViewColumnDidMoveNotification to the default notification center.

noteNumberOfRowsChanged

– (void)**noteNumberOfRowsChanged**

Informs the receiver that the number of records in its data source has changed, allowing the receiver to update the scrollers in its NSScrollView without actually reloading data into the receiver. It's useful for a data source that continually receives data in the background over a period of time, in which case the NSTableView can remain responsive to the user while the data is received.

See the NSTableDataSource informal protocol specification for information on the messages an NSTableView sends to its data source.

See also: – **reloadData**, – **numberOfRowsInTableView:** (NSTableDataSource informal protocol)

numberOfColumns

– (int)**numberOfColumns**

Returns the number of columns in the receiver.

See also: – **numberOfRows**

numberOfRows

– (int)**numberOfRows**

Returns the number of rows in the receiver.

See also: – **numberOfColumns**, – **numberOfRowsInTableView:** (NSTableDataSource informal protocol)

numberOfSelectedColumns

– (int)**numberOfSelectedColumns**

Returns the number of selected columns.

See also: – **numberOfSelectedRows**, – **selectedColumnEnumerator**

numberOfSelectedRows

– (int)**numberOfSelectedRows**

Returns the number of selected rows.

See also: – **numberOfSelectedColumns**, – **selectedRowEnumerator**

rectOfColumn:

– (NSRect)**rectOfColumn:**(int)*columnIndex*

Returns the rectangle containing the column at *columnIndex*. Raises an NSInternalInconsistencyException if *columnIndex* lies outside the range of valid column indices for the NSTableView.

See also: – **frameOfCellAtColumn:row:**, – **rectOfRow:**, – **headerRectOfColumn:** (NSTableHeaderView)

rectOfRow:

– (NSRect)**rectOfRow:**(int)*rowIndex*

Returns the rectangle containing the row at *rowIndex*. Raises an NSInternalInconistencyException if *columnIndex* lies outside the range of valid column indices for the receiver.

See also: – **frameOfCellAtColumn:row:**, – **rectOfColumn:**

reloadData

– (void)**reloadData**

Marks the receiver as needing redisplay, so that it will reload the data for visible cells and draw the new values.

See also: – **noteNumberOfRowsChanged**

removeTableColumn:

– (void)**removeTableColumn:**(NSTableColumn *)*aTableColumn*

Deletes *aTableColumn* from the receiver.

See also: – **sizeLastColumnToFit**, – **sizeToFit**, – **addTableColumn:**

rowAtPoint:

– (int)**rowAtPoint:**(NSPoint)*aPoint*

Returns the index of the row that *aPoint* lies in, or –1 if *aPoint* lies outside the receiver's bounds.

See also: – **columnAtPoint:**

rowHeight

– (float)**rowHeight**

Returns the height of each row in the receiver. The default row height is 16.0.

See also: – **setRowHeight:**

rowsInRect:

– (NSRange)**rowsInRect:**(NSRect)*aRect*

Returns a range of indices for the rows that lie wholly or partially within the vertical boundaries of *aRect*; the location of the range is the first such row's index, and the length is the number of rows that lie in *aRect*. Both the width and height of *aRect* must be nonzero values, or **columnsInRect:** returns an NSRange whose length is zero.

See also: – **columnsInRect:**

scrollColumnToVisible:

– (void)**scrollColumnToVisible:**(int)*columnIndex*

Scrolls the receiver and header view horizontally in an enclosing NSClipView so that the column specified by *columnIndex* is visible.

See also: – **scrollRowToVisible:**, – **scrollToPoint:** (NSClipView)

scrollRowToVisible:

– (void)**scrollRowToVisible:**(int)*rowIndex*

Scrolls the receiver vertically in an enclosing NSClipView so that the row specified by *rowIndex* is visible.

See also: – **scrollColumnToVisible:**, – **scrollToPoint:** (NSClipView)

selectAll:

– (void)**selectAll:**(id)*sender*

If the table allows multiple selection, selects all rows or all columns, according to whether rows or columns were most recently selected. If nothing has been recently selected, this method selects all rows. If this table doesn't allow multiple selection, this method does nothing.

If the selection does change, this method posts NSTableViewSelectionDidChangeNotification to the default notification center.

As a target-action method, **selectAll:** checks with the delegate before changing the selection.

See also: – **allowsMultipleSelection**, – **deselectAll:**, – **selectColumn:byExtendingSelection:**

selectColumn:byExtendingSelection:

– (void)**selectColumn:**(int)*columnIndex* **byExtendingSelection:**(BOOL)*flag*

Selects the column at *columnIndex*, regardless of whether column selection is allowed. If flag is NO, deselects all before selecting the new column. Raises an NSInternalInconistencyException if multiple selection isn't allowed and *flag* is YES. Posts NSTableViewSelectionDidChangeNotification to the default notification center if the selection does in fact change.

This method doesn't check with the delegate before changing the selection. If the user is editing a cell, editing is simply forced to end and the selection is changed.

See also: – **allowsMultipleSelection**, – **allowsColumnSelection**, – **deselectColumn:**, – **selectedColumn**, – **selectRow:byExtendingSelection:**

selectedColumn

– (int)**selectedColumn**

Returns the index of the last column selected or added to the selection, or –1 if no column is selected.

See also: – **selectedColumnEnumerator**, – **numberOfSelectedColumns**, – **selectColumn: byExtendingSelection:**, – **deselectColumn:**

selectedColumnEnumerator

– (NSEnumerator *)**selectedColumnEnumerator**

Returns an object that enumerates the indices of the selected columns as NSNumbers.

See also: – **numberOfSelectedColumns**, – **selectedColumn**, – **selectedRowEnumerator**

selectedRow

– (int)**selectedRow**

Returns the index of the last row selected or added to the selection, or –1 if no row is selected.

See also: – **selectedRowEnumerator**, – **numberOfSelectedRows**, – **selectRow:byExtendingSelection:**, – **deselectRow:**

selectedRowEnumerator

– (NSEnumerator *)**selectedRowEnumerator**

Returns an object that enumerates the indices of the selected rows as NSNumbers.

See also: – **numberOfSelectedRows**, – **selectedRow**, – **selectedColumnEnumerator**

selectRow:byExtendingSelection:

– (void)**selectRow:**(int)*rowIndex* **byExtendingSelection:**(BOOL)*flag*

Selects the row at *rowIndex*. If flag is NO, deselects all before selecting the new row. Raises an NSInternalInconistencyException if multiple selection isn't allowed and flag is YES. Posts NSTableViewSelectionDidChangeNotification to the default notification center if the selection does in fact change.

This method doesn't check with the delegate before changing the selection. If the user is editing a cell, editing is simply forced to end and the selection is changed.

See also: – **allowsMultipleSelection**, – **deselectRow:**, – **selectedRow**, – **selectColumn: byExtendingSelection:**

setAllowsColumnReordering:

– (void)**setAllowsColumnReordering:**(BOOL)*flag*

Controls whether the user can drag column headers to reorder columns. If *flag* is YES the user can reorder columns; if *flag* is NO the user can't. The default is YES. You can rearrange columns programmatically regardless of this setting.

See also: – **moveColumn:toColumn:**, – **allowsColumnReordering**

setAllowsColumnResizing:

– (void)**setAllowsColumnResizing:**(BOOL)*flag*

Controls whether the user can resize columns by dragging between headers. If *flag* is YES the user can resize columns; if *flag* is NO the user can't. The default is YES. You can resize columns programmatically regardless of this setting.

See also: – **setWidth:** (NSTableColumn), – **allowsColumnResizing**

setAllowsColumnSelection:

– (void)**setAllowsColumnSelection:**(BOOL)*flag*

Controls whether the user can select an entire column by clicking its header. If *flag* is YES the user can select columns; if *flag* is NO the user can't. The default is YES. You can select columns programmatically regardless of this setting.

See also: – **selectColumn:byExtendingSelection:**, – **setAllowsColumnReordering:**, – **allowsColumnSelection**

setAllowsEmptySelection:

– (void)**setAllowsEmptySelection:**(BOOL)*flag*

Controls whether the receiver allows zero rows or columns to be selected. If *flag* is YES empty selection is allowed; if *flag* is NO it isn't. The default is YES.

You can *not* set an empty selection programmatically if empty selection is disallowed, unlike with the other settings that affect selection behavior.

See also: – **deselectAll:**, – **deselectColumn:**, – **deselectRow:**, – **allowsEmptySelection**

setAllowsMultipleSelection:

– (void)**setAllowsMultipleSelection:**(BOOL)*flag*

Controls whether the user can select more than one row or column at a time. If *flag* is YES the user can select multiple rows or columns; if *flag* is NO the user can't. The default is NO. You can select multiple columns or rows programmatically regardless of this setting.

See also: – **selectColumn:byExtendingSelection:**, – **selectRow:byExtendingSelection:**, – **allowsMultipleSelection**

setAutoresizesAllColumnsToFit:

– (void)**setAutoresizesAllColumnsToFit:**(BOOL)*flag*

Controls whether the receiver proportionally resizes its columns to fit when its superview's frame changes. If *flag* is YES, the difference in width is distributed among the receiver's table columns; if *flag* is NO, only the last column is resized to fit.

See also: – **autoresizesAllColumnsToFit**, – **sizeLastColumnToFit**, – **sizeToFit**

setAutosaveName:

– (void)**setAutosaveName**:(NSString *)*name*

Sets the name under which table information is automatically saved to *name*. If *name* is different from the current name, this method also reads in the saved information and sets the order and width of this table view's columns to match.

The table information is saved separately for each user and for each application that user uses. Note that even though a table view has an autosave name, it may not be saving table information automatically . To set whether table information is being saved automatically, use **setAutosaveTableColumns:**.

See also: **autosaveName**, **setAutosaveTableColumns:**

setAutosaveTableColumns:

– (void)**setAutosaveTableColumns**:(BOOL)*flag*

Sets whether the order and width of this table view's columns are automatically saved. If *flag* is different from the current value, this method also reads in the saved information and sets the table options to match.

The table information is saved separately for each user and for each application that user uses. Note that if **autosaveName** returns **nil**, this setting is ignored and table information isn't saved.

See also: **autosaveTableColumns**, **setAutosaveName:**

setBackgroundColor:

– (void)**setBackgroundColor:**(NSColor *)*aColor*

Sets the receiver's background color to *aColor*.

See also: – **setNeedsDisplay:** (NSView), – **backgroundColor**

setCornerView:

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

Sets the receiver's corner view to *aView*. The default corner view merely draws a bezeled rectangle using a blank NSTableHeaderCell, but you can replace it with a custom view that displays an image or with a control that can handle mouse events, such as a select-all button. Your custom corner view should be as wide as a vertical NSScroller and as tall as the receiver's header view.

See also: – **setHeaderView:**, – **cornerView**

setDataSource:

– (void)**setDataSource:**(id)*anObject*

Sets the receiver's data source to *anObject* and invokes **tile**. *anObject* should implement the appropriate methods of the NSTableDataSource informal protocol.

This method raises an NSInternalInconistencyException if *anObject* doesn't respond to either **numberOfRowsInTableView:** or **tableView:objectValueForTableColumn:row:**.

See also: – **dataSource**

setDelegate:

– (void)**setDelegate:**(id)*anObject*

Sets the receiver's delegate to *anObject*.

See also: – **delegate**

setDoubleAction:

– (void)**setDoubleAction:**(SEL)*aSelector*

Sets to *aSelector* the message sent to the target when the user double-clicks an uneditable cell or a column header. If the double-clicked cell is editable, this message isn't sent and the cell is edited instead. You can use this method to implement features such as sorting records according to the column that was double-clicked.

See also: – **setAction:** (NSControl), – **setTarget:** (NSControl), – **doubleAction**

setDrawsGrid:

– (void)**setDrawsGrid:**(BOOL)flag

Controls whether the receiver draws grid lines around cells. If *flag* is YES it does; if flag is NO it doesn't. The default is YES.

See also: – **setGridColor:**, – **drawGridInClipRect:**, – **drawsGrid**

setGridColor:

– (void)**setGridColor:**(NSColor *)*aColor*

Sets the color used to draw grid lines to *aColor*. The default color is gray.

See also: – **setDrawsGrid:**, – **drawGridInClipRect:**, – **gridColor**

setHeaderView:

– (void)**setHeaderView:**(NSTableHeaderView *)*aHeaderView*

Sets the receiver's header view to *aHeaderView*.

See also: – **setCornerView:**, – **headerView**

setIntercellSpacing:

– (void)**setIntercellSpacing:**(NSSize)*aSize*

Sets the width and height between cells to those in *aSize* and redisplays the receiver. The default intercell spacing is (3.0, 2.0).

See also: – **intercellSpacing**

setRowHeight:

– (void)**setRowHeight:**(float)*rowHeight*

Sets the height for rows to *rowHeight* and invokes **tile**.

See also: – **rowHeight**

sizeLastColumnToFit

– (void)**sizeLastColumnToFit**

Resizes the last column if there's room so that the receiver fits exactly within its enclosing NSClipView.

See also: – **setAutoresizesAllColumnsToFit:**, – **minWidth** (NSTableColumn),
– **maxWidth** (NSTableColumn)

sizeToFit

– (void)**sizeToFit**

Resizes columns if there's room so that all fit in the enclosing NSClipView and so all but the last are just wide enough to display their titles and values. This method first sets all columns to their minimum widths; then divides among the columns the space remaining to fill the width of the NSScrollView.

See also: – **setAutoresizesAllColumnsToFit:**, – **minWidth** (NSTableColumn), – **maxWidth** (NSTableColumn)

tableColumns

– (NSArray *)**tableColumns**

Returns the NSTableColumns in the receiver.

tableColumnWithIdentifier:

– (NSTableColumn *)**tableColumnWithIdentifier:**(id)*anObject*

Returns the NSTableColumn object for the first column whose identifier is equal to *anObject*, as compared using **isEqual:**, or **nil** if no columns are found with the specified identifier.

See also: – **columnWithIdentifier:**

textDidBeginEditing:

– (void)**textDidBeginEditing:**(NSNotification *)*aNotification*

Posts an NSControlTextDidBeginEditingNotification to the default notification center, as described in the NSControl class specification. *aNotification* is the NSNotification posted by the field editor; see the NSText class specifications for more information on this text delegate method.

See also: – **textShouldBeginEditing:**

textDidChange:

– (void)**textDidChange:**(NSNotification *)*aNotification*

Sends **textDidChange:** to the edited cell, and posts an NSControlTextDidChangeNotification to the default notification center, as described in the NSControl class specification. *aNotification* is the NSNotification posted by the field editor; see the NSText class specifications for more information on this text delegate method.

textDidEndEditing:

– (void)**textDidEndEditing:**(NSNotification *)*aNotification*

Updates the data source based on the newly-edited value and selects another cell for editing if possible according to the character that ended editing (Return, Tab, Backtab). *aNotification* is the NSNotification posted by the field editor; see the NSText class specifications for more information on this text delegate method.

See also: – **textShouldEndEditing:**

textShouldBeginEditing:

– (BOOL)**textShouldBeginEditing:**(NSNotification *)*aNotification*

Queries the delegate using **control:textShouldBeginEditing:**, returning the delegate's response, or simply returning YES to allow editing if the delegate doesn't respond to that method. *aNotification* is the NSNotification posted by the field editor; see the NSText class specifications for more information on this text delegate method.

See also: – **textDidBeginEditing:**

textShouldEndEditing:

– (BOOL)**textShouldEndEditing:**(NSNotification *)*aNotification*

Validates the cell being edited and queries the delegate using **control:textShouldEndEditing:**, returning the delegate's response if it responds to that method. If it doesn't, it returns YES if the cell's new value is valid and NO if it isn't. *aNotification* is the NSNotification posted by the field editor; see the NSText class specifications for more information on this text delegate method.

See also: – **textDidEndEditing:**

tile

– (void)**tile**

Properly sizes the receiver and its header view, and marks it as needing display. Also resets cursor rectangles for the header view and line scroll amounts for the NSScrollView.

See also: – **setNeedsDisplay:** (NSView)

Methods Implemented By the Delegate

selectionShouldChangeInTableView:

– (BOOL)**selectionShouldChangeInTableView:**(NSTableView *)*aTableView*

Returns YES to permit *aTableView* to change its selection (typically a row being edited), NO to deny permission. The user can select and edit different cells within the same row, but can't select another row unless the delegate approves. The delegate can implement this method for complex validation of edited rows based on the values of any of their cells.

tableView:shouldEditTableColumn:row:

– (BOOL)**tableView:**(NSTableView *)*aTableView*
shouldEditTableColumn:(NSTableColumn *)*aTableColumn*
row:(int)*rowIndex*

Returns YES to permit *aTableView* to edit the cell at *rowIndex* in *aTableColumn*, NO to deny permission. The delegate can implement this method to disallow editing of specific cells.

tableView:shouldSelectRow:

– (BOOL)**tableView:**(NSTableView *)*aTableView*
shouldSelectRow:(int)*rowIndex*

Returns YES to permit *aTableView* to select the row at *rowIndex*, NO to deny permission. The delegate can implement this method to disallow selection of particular rows.

tableView:shouldSelectTableColumn:

– (BOOL)**tableView:**(NSTableView *)*aTableView*
shouldSelectTableColumn:(NSTableColumn *)*aTableColumn*

Returns YES to permit *aTableView* to select *aTableColumn*, NO to deny permission. The delegate can implement this method to disallow selection of particular columns.

tableView:willDisplayCell:forTableColumn:row:

– (void)**tableView:**(NSTableView *)*aTableView*
willDisplayCell:(id)*aCell*
forTableColumn:(NSTableColumn *)*aTableColumn*
row:(int)*rowIndex*

Informs the delegate that *aTableView* will display the cell at *rowIndex* in *aTableColumn* using *aCell*. The delegate can modify the display attributes of *aCell* to alter the appearance of the cell. Since *aCell* is reused for every row in *aTableColumn*, the delegate must set the display attributes both when drawing special cells and when drawing normal cells.

tableViewColumnDidMove:

– (void)**tableViewColumnDidMove:**(NSNotification *)*aNotification*

Informs the delegate that a column was moved by user action in the NSTableView. *aNotification* is an NSTableViewColumnDidMoveNotification.

tableViewColumnDidResize:

– (void)**tableViewColumnDidResize:**(NSNotification *)*aNotification*

Informs the delegate that a column was resized in the NSTableView. *aNotification* is an NSTableViewColumnDidResizeNotification.

tableViewSelectionDidChange:

– (void)**tableViewSelectionDidChange:**(NSNotification *)*aNotification*

Informs the delegate that the NSTableView's selection has changed. *aNotification* is an NSTableViewSelectionDidChangeNotification.

tableViewSelectionIsChanging:

– (void)**tableViewSelectionIsChanging:**(NSNotification *)*aNotification*

Informs the delegate that the NSTableView's selection is in the process of changing (typically because the user is dragging the mouse across a number of rows). *aNotification* is an NSTableViewSelectionIsChangingNotification.

Notifications

NSTableViewColumnDidMoveNotification

Posted whenever a column is moved by user action in the NSTableView. The notification object is the NSTableView in which a column moved. The userInfo dictionary contains these keys and values:

Key	Value
NSOldColumn	The column's original index (an NSNumber)
NSNewColumn	The column's present index (an NSNumber)

See also: – **moveColumn:toColumn:**

NSTableViewColumnDidResizeNotification

Posted whenever a column is resized in the NSTableView. The notification object is the NSTableView in which a column was resized. The userInfo dictionary contains these keys and values:

Key	Value
NSOldWidth	The column's original width (an NSNumber)

NSTableViewSelectionDidChangeNotification

Posted after the NSTableView's selection changes. The notification object is the NSTableView whose selection changed. The userInfo dictionary is **nil**.

NSTableViewSelectionIsChangingNotification

Posted as the NSTableView's selection changes (while the mouse is still down). The notification object is the NSTableView whose selection is changing. The userInfo dictionary is **nil**.

NSTabView

Inherits From: NSView : NSResponder : NSObject

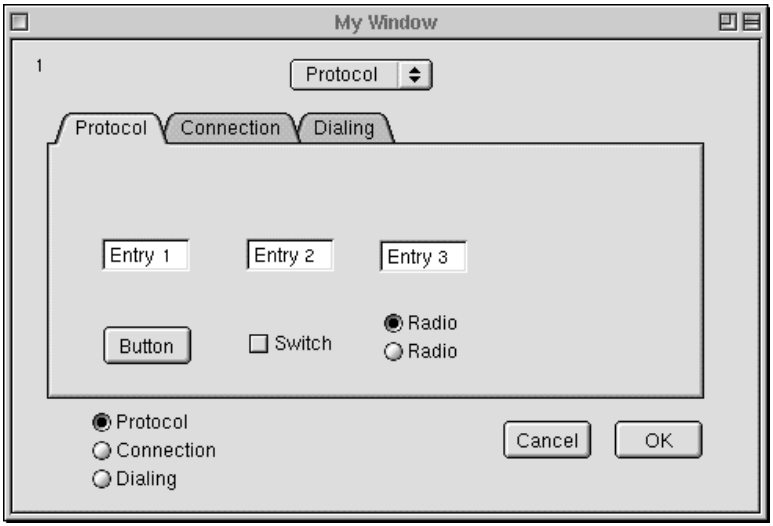
Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSTabView.h

Class Description

Note: The NSTabView class and its supporting class NSTabViewItem are under development. If you want to use these classes, you will have to instantiate them programmatically because Interface Builder does not yet include support for them.

An NSTabView provides a convenient mechanism for presenting information in a multi-page format. The view is distinguished by a row of tabs that give the visual appearance of folder tabs, as shown in the figure below. The user selects the desired page by clicking the appropriate tab or using the arrow keys to move between pages. Each page displays a view hierarchy provided by your application.



An NSTabView can support a multi-page format without visible tabs. For example, instead of tabs, you might use a pop-up menu or radio buttons, similar to those shown in the illustration, to let the user select from several view pages. When a tab view is drawn with tabs (the default), the border must be bezeled. When a tab view is drawn without tabs, the view can have a bezeled border, a lined border, or no border.

An NSTabView keeps a zero-based array of NSTabViewItems, one per tab in the view. A tab view item provides access to a tab's color, state, label text, initial first responder, and associated view.Your application can supply each tab view item with an optional identifier object to customize tab handling. For more information, see the documentation for NSTabViewItem.Tab label text defaults to the default font and font size used for standard interface items, such as button labels and menu items. When you invoke **setFont:** to change the tab view's font, tab height and width is adjusted automatically to accommodate a new font size. If the view allows truncating, tab labels are truncated as needed.

Delegate Messages

NSTabView defines delegate messages to allow the delegate to control or react to changes in selection and changes in the number of tabs:

tabViewDidChangeNumberOfTabViewItems: informs the delegate that the number of tab view items in *tabView* has changed.

tabView:didSelectTabViewItem: informs the delegate that the specified tab view item has been selected.

tabView:shouldSelectTabViewItem: informs the delegate that the specified tab view item is about to be selected. The delegate can return NO to prevent the selection.

tabView:willSelectTabViewItem: informs the delegate that the specified tab view item will be selected. The delegate can perform tasks related to the selection, but cannot prevent it.

Method Types

Adding and removing tabs

– addTabViewItem:
– insertTabViewItem: atIndex:
– removeTabViewItem:

Accessing tabs

– indexOfTabViewItem:
– indexOfTabViewItemWithIdentifier:
– numberOfTabViewItems
– tabViewItemAtIndex:
– tabViewItems

Selecting a tab

– selectFirstTabViewItem:
– selectLastTabViewItem:
– selectNextTabViewItem:
– selectPreviousTabViewItem:
– selectedTabViewItem
– selectTabViewItemAtIndex:
– takeSelectedTabViewItemFromSender:

Modifying the font

– font
– setFont:

Modifying the tab type

– setTabViewType:
– tabViewType

Manipulating the background

– drawsBackground
– setDrawsBackground:

Determining the size

– minimumSize
– contentRect

Truncating tab labels

– allowsTruncatedLabels
– setAllowsTruncatedLabels:

Assigning a delegate

– setDelegate:
– delegate

Event handling

– tabViewItemAtPoint:

Instance Methods

addTabViewItem:

– (void)**addTabViewItem:**(NSTabViewItem *)*tabViewItem*

Adds the tab item specified by *tabViewItem*. The item is added at the end of the array of tab items, so that the new tab appears on the right side of the view. If the delegate supports it, invokes the delegate's **tabViewDidChangeNumberOfTabViewItems:** method.

See also: – **insertTabViewItem:atIndex:**, – **numberOfTabViewItems**, – **removeTabViewItem:**, – **tabViewItemAtIndex:**, – **tabViewItems**

allowsTruncatedLabels

– (BOOL)**allowsTruncatedLabels**

Returns YES if the tab view allows truncating for labels that don't fit on a tab. The default is NO. When truncating is allowed, the tab view inserts an ellipsis, if necessary, to fit a label in the tab.

See also: – **setAllowsTruncatedLabels:**

contentRect

– (NSRect)**contentRect**

Returns the rectangle describing the content area of a tab view. This area does not include the space required for the tab view's tabs or borders (if any).

delegate

– (id)**delegate**

Returns the tab view's delegate.

See also: – **setDelegate:**

drawsBackground

– (BOOL) **drawsBackground**

Returns YES if the tab view draws a background color when the tab view type is NSNoTabsNoBorder. If the tab view uses bezeled edges or a line border, the appropriate background color for that border is used.

See also: **– setTabViewType:**, **– setDrawsBackground:**

font

– (NSFont *)**font**

Returns the font for tab label text.

See also: **– setFont:**

indexOfTabViewItem:

– (int)**indexOfTabViewItem**:(NSTabViewItem *)*tabViewItem*

Returns the index of the item that matches *tabViewItem*, or NSNotFound if the item is not found. A tab view keeps an array containing one tab view item for each tab in the view—this is the array that is searched. The returned index is base 0.

See also: **– indexOfTabViewItemWithIdentifier:**, **– insertTabViewItem:atIndex:**, **– numberOfTabViewItems**, **– tabViewItemAtIndex:**

indexOfTabViewItemWithIdentifier:

– (int)**indexOfTabViewItemWithIdentifier**:(id)*identifier*

Returns the index of the item that matches *identifier*, or NSNotFound if the item is not found. A tab view keeps an array containing one tab view item for each tab in the view—this is the array that is searched. The returned index is base 0.

See also: **– indexOfTabViewItem:**, **– insertTabViewItem:atIndex:**, **– numberOfTabViewItems**, **– tabViewItemAtIndex:**

insertTabViewItem:atIndex:

– (void)**insertTabViewItem:**(NSTabViewItem *)*tabViewItem*
atIndex:(int)*index*

Inserts *tabViewItem* into the tab view's array of tab view items at *index*. The *index* parameter is base 0. If there is a delegate and the delegate supports it, sends the delegate the **tabViewDidChangeNumberOfTabViewItems:** message.

See also: – **indexOfTabViewItem:**, – **indexOfTabViewItemWithIdentifier:**, – **numberOfTabViewItems**, – **tabViewItemAtIndex:**

minimumSize

– (NSSize)**minimumSize**

Returns the minimum size necessary for the view to display tabs in a useful way. You can use the value returned by this method to limit how much a user can resize a tab view.

See also: – **setTabViewType:**

numberOfTabViewItems

– (int)**numberOfTabViewItems**

Returns the number of items in the tab view's array of tab view items. Because there is one item in the array for each tab in the view, this is equivalent to the number of tabs in the view.

See also: – **indexOfTabViewItem:**, – **tabViewItems**

removeTabViewItem:

– (void)**removeTabViewItem:**(NSTabViewItem *)*tabViewItem*

Removes the item specified by *tabViewItem* from the tab view's array of tab view items. If there is a delegate and the delegate supports it, sends the delegate the **tabViewDidChangeNumberOfTabViewItems:** message.

See also: – **addTabViewItem:**, – **insertTabViewItem:atIndex:**, – **tabViewItems**

selectFirstTabViewItem:

– (void)**selectFirstTabViewItem:**(id)*sender*

Selects the first tab view item. The *sender* parameter is ignored.

See also: **– selectTabViewItem:**

selectLastTabViewItem:

– (void)**selectLastTabViewItem:**(id)*sender*

Selects the last tab view item. The *sender* parameter is ignored.

See also: **– selectTabViewItem:**

selectNextTabViewItem:

– (void)**selectNextTabViewItem:**(id)*sender*

Selects the next tab view item in the sequence. If the currently visible item is the last item in the sequence, this method does nothing and the last page remains displayed. The *sender* parameter is ignored.

See also: **– selectTabViewItem:**

selectPreviousTabViewItem:

– (void)**selectPreviousTabViewItem:**(id)*sender*

Selects the previous tab view item in the sequence. If the currently visible item is the first item in the sequence, this method does nothing and the first page remains displayed. The *sender* parameter is ignored.

See also: **– selectTabViewItem:**

selectedTabViewItem

– (NSTabViewItem *)**selectedTabViewItem**

Returns the tab view item for the currently-selected tab, or **nil** if no item is selected. If there is a delegate and the delegate supports it, sends the delegate the **shouldSelectTabViewItem:** message.

See also: **– selectTabViewItemAtIndex:**

selectTabViewItem:

– (void)**selectTabViewItem:**(NSTabViewItem *)*tabViewItem*

Selects the tab view item specified by *tabViewItem*. If there is a delegate and the delegate supports it, sends the delegate the **shouldSelectTabViewItem:** message.

See also: – **insertTabViewItem:atIndex:**, – **selectedTabViewItem**

selectTabViewItemAtIndex:

– (void)**selectTabViewItemAtIndex:**(int)*index*

Selects the tab view item specified by *index*. The *index* parameter is base 0.

See also: – **insertTabViewItem:atIndex:**, – **selectedTabViewItem**

selectTabViewItemWithIdentifier

– (void)**selectTabViewItemWithIdentifier:**(id)*identifier*;

Selects the tab view item specified by *identifier*.

See also: – **setIdentifier:** (NSTabViewItem), – **identifier** (NSTabViewItem),
– **selectTabViewItemAtIndex:**, – **selectedTabViewItem**

setAllowsTruncatedLabels:

– (void)**setAllowsTruncatedLabels:**(BOOL)*allowTruncatedLabels*

Returns YES if the tab view allows truncating for names that don't fit on a tab.

See also: – **allowsTruncatedLabels**

setDelegate:

– (void)**setDelegate:**(id)*anObject*

Sets the tab view's delegate to *anObject*.

See also: – **delegate**

setDrawsBackground:

– (void)**setDrawsBackground:**(BOOL)*flag*

Sets whether a background is drawn when the view type is NSNoTabsNoBorder. If the tab view has a bezeled border or a line border the appropriate background for that border is used.

See also: **– setTabViewType:**, **– drawsBackground**

setFont:

– (void)**setFont:**(NSFont *)*font*

Sets the font for tab label text to *font*. Tab height is adjusted automatically to accommodate a new font size. If the view allows truncating, tab labels are truncated as needed.

See also: – **allowsTruncatedLabels**, – **font**, – **setAllowsTruncatedLabels:**

setTabViewType:

– (void)**setTabViewType:**(NSTabViewType)*tabViewType*

Sets the tab type to *tabViewType*. The available types are:

Tab Type	Description
NSTopTabsBezelBorder	The view includes tabs and has a bezeled border (the default)
NSNoTabsBezelBorder	The view does not include tabs and has a bezeled border
NSNoTabsLineBorder	The view does not include tabs and has a lined border
NSNoTabsNoBorder	The view does not include tabs and has no border

See also: – **tabViewType**

tabViewItemAtIndex:

– (NSTabViewItem *)**tabViewItemAtIndex:**(int)*index*

Returns the tab view item at *index* in the tab view's array of items. The *index* parameter is base 0.

See also: – **indexOfTabViewItem:**, – **insertTabViewItem:atIndex:**, – **tabViewItems**

tabViewItemAtPoint:

– (NSTabViewItem *)**tabViewItemAtPoint:**(NSPoint)*point*

Returns the tab view item identified by *point*. You can use this method to find a tab view item based on a user's mouse click.

tabViewItems

– (NSArray *)**tabViewItems**

Returns the tab view's array of tab view items. A tab view keeps an array containing one tab view item for each tab in the view. The array is base 0.

See also: – **numberOfTabViewItems**, – **tabViewItemAtIndex:**

tabViewType

– (NSTabViewType)**tabViewType**

Returns the tab type for the tab view. The available types are described with the **setTabViewType:** method.

takeSelectedTabViewItemFromSender:

– (void)**takeSelectedTabViewItemFromSender:**(id)*sender*

If *sender* responds to the **indexOfSelectedItem** method, this method invokes that method and selects the tab view item at the specified index. If *sender* does not respond to **indexOfSelectedItem** but is an instance of NSMatrix, this method uses the index of the matrix' currently selected cell.

The location of the selected cell is a zero-based number, obtained by counting the number of cells up to and including the selected cell. Cells are counted from left to right and from top to bottom. For example in a 5 by 5 matrix, if the selected cell is three rows down in column five (location [2,4] in the matrix), the corresponding index would be 14.

Methods Implemented By the Delegate

tabViewDidChangeNumberOfTabViewItems:

– (void)**tabViewDidChangeNumberOfTabViewItems:**(NSTabView *)*tabView*

Informs the delegate that the number of tab view items in *tabView* has changed.

See also: – **numberOfTabViewItems** (NSTabView class)

tabView:didSelectTabViewItem:

– (void)**tabView:**(NSTabView *)*tabView*
didSelectTabViewItem:(NSTabViewItem *)*tabViewItem*

Informs the delegate that *tabView* has selected *tabViewItem*.

tabView:shouldSelectTabViewItem:

– ((BOOL)**tabView:**(NSTabView *)*tabView*
shouldSelectTabViewItem:(NSTabViewItem *)*tabViewItem*

Invoked just before *tabViewItem* in *tabView* is selected. The delegate can return NO to prevent selection of specific tabs.

tabView:willSelectTabViewItem:

– (void)**tabView:**(NSTabView *)*tabView*
willSelectTabViewItem:(NSTabViewItem *)*tabViewItem*

Informs the delegate that *tabView* is about to select *tabViewItem*.

NSTabViewItem

Inherits From: NSObject

Conforms To: NSCoding
NSObject (NSObject)

Declared In: AppKit/NSTabViewItem.h

Class Description

Note: The NSTabView class and its supporting class NSTabViewItem are under development. If you want to use these classes, you will have to instantiate them programmatically because Interface Builder does not yet include support for them. An NSTabView provides a convenient mechanism for presenting information in a multi-page format. A tab view is usually distinguished by a row of tabs that give the visual appearance of folder tabs. When the user clicks on a tab, the tab view displays a view page provided by your application.

A tab view keeps a zero-based array of NSTabViewItems, one for each tab in the view. A tab view item provides access to the tab's color, state, label text, initial first responder, and associated view. Your application can supply each tab view item with an optional identifier object to customize tab handling.

Method Types

Creating a tab view item
– initWithIdentifier:

Working with labels
– drawLabel:inRect:
– label
– setLabel:
– sizeOfLabel:

Checking the tab display state
– tabState

Assigning an identifier object
– identifier
– setIdentifier:

Setting the color

– color
– setColor:

Assigning a view

– view
– setView:

Setting the initial first responder

– initialFirstResponder
– setInitialFirstResponder:

Accessing the parent tab view

– tabView

Instance Methods

color

– (NSColor *)**color**

Returns the color for the tab view item. By default, the color is set to the system color used for the flat surfaces of a control.

See also: – **setColor:**

drawLabel:inRect:

– (void)**drawLabel:**(BOOL)*shouldTruncateLabel* **inRect:(**NSRect)*tabRect*

If *shouldTruncateLabel* is NO, draws the full label in the rect specified by *tabRect*. If *shouldTruncateLabel* is YES, draws the truncated label. You can override this method to perform customized label drawing. For example, you might want to add an icon to each tab in the view.

See also: – **sizeOfLabel:**

identifier

– (id)**identifier**

Returns the tab view item's optional identifier object. To customize how your application works with tabs, you can initialize each tab view item with an identifier object.

See also: – **initWithIdentifier**:, – **setIdentifier:**

initialFirstResponder

– (id)**initialFirstResponder**

Returns the **id** for the initial first responder for the view associated with the tab view item.

See also: – **setInitialFirstResponder:**

initWithIdentifier:

– (id)**initWithIdentifier:**(id)*identifier*

Performs default initialization for the tab view item. Sets the item's identifier object to *identifier*, if it is not **nil**. Use this method when creating tab view items programmatically.

See also: – **identifier**, – **setIdentifier:**

label

– (NSString *)**label**

Returns the label text for the tab view item.

See also: – **setLabel:**

setColor:

– (void)**setColor:**(NSColor *)*color*

Sets the color for the tab view item to *color*.

See also: – **color**

setIdentifier:

– (void)**setIdentifier:**(id)*identifier*

Sets the tab view item's optional identifier object to *identifier*. To customize how your application works with tabs, you can specify an identifier object for each tab view item.

See also: – **identifier**, – **initWithIdentifier:**

setInitialFirstResponder:

– (void)**setInitialFirstResponder:**(NSView *)*view*

Sets the initial first responder for the view associated with the tab view item (the view that is displayed when a user clicks on the tab) to *view*.

See also: – **initialFirstResponder**

setLabel:

– (void)**setLabel:**(NSString *)*label*

Sets the label text for the tab view item according to the passed string *label*.

See also: – **label**

setView:

– (void)**setView:**(NSView *)*view*

Sets the view associated with the tab view item to *view*. This is the view that is displayed when a user clicks on the tab. When you set a new view, the old view is released.

See also: – **view**

sizeOfLabel:

– (NSSize)**sizeOfLabel:**(BOOL)*shouldTruncateLabel*

If *shouldTruncateLabel* is NO, returns the size of the tab view item's full label. If *shouldTruncateLabel* is YES, returns the truncated size. If your application does anything to change the size of tab labels, such as overriding the **drawLabel:inRect:** method to add an icon to each tab, you should override **sizeOfLabel:** too so that the NSTabView knows the correct size for the tab label.

See also: – **drawLabel:inRect:**, – **setFont**: (NSTabView)

tabState

– (NSTabState)**tabState**

Returns the current display state of the tab associated with this tab view item. The possible values are NSSelectedTab, NSBackgroundTab, or NSPressedTab. Your application does not directly set the tab state.

tabView

– (NSTabView *)**tabView**

Returns the parent tab view for the tab view item. Note that this is the tab view itself, not the view that is displayed when a user clicks on the tab.

A tab view item normally learns about its parent tab view when it is inserted into the view's array of items. The NSTabView methods **addTabViewItem:** and **insertTabViewItem:atIndex:** set the tab view for the added or inserted item.

See also: – **setView**:, – **view**

view

– (id)**view**

Returns the **id** for the view associated with the tab view item. This is the view that is displayed when a user clicks on the tab.

See also: – **setView:**

NSText

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

Class at a Glance

Purpose

NSText declares the most general programmatic interface for objects that manage text. You usually use instances of its subclass, NSTextView.

Principal Attributes

Supports rich text and graphics	Provides delegation and notification
Works with the Font Panel and menu	Works with the pasteboard
Works with the Services facility	Works with the spell-checking service

Creation

See the class description.

Commonly Used Methods

– readRTFDFromFile:	Reads an **.rtf** or **.rtfd** file.
– writeRTFDToFile:atomically:	Writes the receiver's text to a file.
– string	Returns the receiver's text, without attributes.
– RTFFromRange:	Returns the receiver's text with attributes.
– RTFDFromRange:	Returns the receiver's text with attributes and attachments.

Class Description

The NSText class declares the most general programmatic interface to objects that manage text. NSText is part of the OpenStep specification. Rhapsody offers a subclass of NSText, NSTextView, that extends the interface declared by NSText and provides much more sophisticated functionality than that declared in NSText.

NSText's **initWithFrame:** method creates an instance of a concrete subclass, such as NSTextView, depending on the platform. Instances of any of these classes are generically called *text objects*.

Text objects are used by the Application Kit wherever text appears in interface objects: A text object draws the title of a window, the commands in a menu, the title of a button, and the items in a browser. Your application can also create text objects for its own purposes.

The text classes are unlike most other classes in the Application Kit in the richness and complexity of their interface. One of their design goals is to provide a comprehensive set of text-handling features so that you'll rarely need to create a subclass. Among other things, a text object can:

- Control whether the user can select or edit text.
- Control the font and layout characteristics of its text by working with the Font Panel and menu.
- Let the user control the format of paragraphs by manipulating a ruler.
- Control the color of its text and background.
- Wrap text on a word or character basis.
- Display graphic images within its text.
- Write text to or read text from files in the form of RTFD—Rich Text Format files that contain TIFF or EPS images, or attached files.
- Let another object, the delegate, dynamically control its properties.
- Let the user copy and paste text within and between applications.
- Let the user copy and paste font and format information between NSText objects.
- Let the user check the spelling of words in its text.

Graphical user-interface building tools (such as Interface Builder) may give you access to text objects in several different configurations, such as those found in the NSTextField, NSForm, and NSScrollView objects. These classes configure a text object for their own specific purposes. Additionally, all NSTextFields, NSForms, NSButtons within the same window—in short, all objects that access a text object through associated cells—share the same text object, reducing the memory demands of an application. Thus, it's generally best to use one of these classes whenever it meets your needs, rather than create text objects yourself. If one of these classes doesn't provide enough flexibility for your purposes, you can create text objects programmatically.

Text objects typically work closely with various other objects. Some of these—such as the delegate or an embedded graphic object—require a degree of programming on your part. Others—such as the Font Panel, spell checker, or ruler—take no effort other than deciding whether the service should be enabled or disabled. Several of the following sections discuss these interrelationships.

Plain and Rich Text Objects

Text objects are differentiated into two groups: those that allow only one set of text attributes for all of their text, and those that allow multiple fonts, sizes, indents, and other attributes for different sets of characters and paragraphs. Text objects in the former group are called *plain* text objects, while those in the latter are

called *rich* text objects. You can control whether a text object is plain or rich using the **setRichText:** method. Rich text objects are also capable of allowing the user to drag images and files into them. This behavior is controlled by the **setImportsGraphics:** method.

A rich NSText object can use RTF (Rich Text Format) as an interchange format. Not all RTF control words are supported: On input, an NSText object ignores any control word it doesn't recognize; some of those it can read and interpret it doesn't write out. The table below lists the RTF control words that any text object recognizes. Subclasses may recognize more.

Control Word	Read	Write
\ansi	yes	yes
\b	yes	yes
\cb	yes	yes
\cf	yes	yes
\colortbl	yes	yes
\dnn	yes	yes
\fin	yes	yes
\fn	yes	yes
\fonttbl	yes	yes
\fsn	yes	yes
\i	yes	yes
\lin	yes	yes
\margrn	yes	yes
\paperwn	yes	yes
\mac	yes	no
\margln	yes	yes
\par	yes	yes
\pard	yes	no
\pca	yes	no

Control Word	Read	Write
\qc	yes	yes
\ql	yes	yes
\qr	yes	yes
\sn	yes	no
\tab	yes	yes
\upn	yes	yes

Notifying a Text Object's Delegate

Many of an NSText object's actions can be controlled through an associated object, the NSText object's delegate. The delegate can be any object you choose, and one delegate can control multiple NSText objects. If it implements any of the following methods, the delegate receives the corresponding message at the appropriate time:

textShouldBeginEditing:
textDidBeginEditing:
textDidChange:
textShouldEndEditing:
textDidEndEditing:

Of special note are the two "textShould" methods. These methods are requests for permission. Any time a text object begins an operation that would change its text or attributes, it uses **textShouldBeginEditing:** to request approval for the change. The delegate can return YES to permit the change, or NO to forbid it. Similarly, **textShouldEndEditing:** enables the delegate to prevent a text object from ending editing, such as when it contains an invalid value.

Adding Graphics and Other Attachments to the Text

A rich text object may allow graphics or other file attachments to be embedded in the text. Each graphic is treated as a single (possibly large) "character": The text's line height and character placement are adjusted to accommodate the graphic. Graphics are embedded in the text in either of two ways: programmatically or directly through user actions. In the programmatic approach, graphic objects can be added using **replaceRange:WithRTFD:**, or through a more specific method defined by a subclass.

An alternate means of adding an image or other attachment to the text is for the user to drag an image or other file directly into the text object. The text object automatically creates an attachment object to manage the display of the image. This feature requires a rich text object that has been configured to receive dragged images using the **setImportsGraphics:** method.

Images that have been imported can be written as part of an RTFD document. RTFD documents use a file package, or directory, to store the components of the document (the “D” stands for “directory”). The file package has the name of the document plus an **.rtfd** extension. The file package always contains a file called **TXT.rtf** for the text of the document, and one or more TIFF or EPS files for the images, plus the files for other attachments. A text object can transfer information in an RTFD document to a file and read it from a file using the **writeRTFDToFile:atomically:** and **readRTFDFromFile:** methods.

Working with the Font Panel

Text objects are designed to work with the Application Kit’s font conversion system, defined by the NSFontPanel and NSFontManager classes. By default, a text object keeps the Font Panel updated with the first font in its selection, or of its typing attributes (defined below). It also changes the font in response to messages from the Font Panel and Font menu. Such changes apply to the selected text or typing attributes for a rich text object, or to all the text in a plain text object. You can turn this behavior off using the **setUsesFontPanel:** method. Doing so is recommended for a text object that serves as a field editor, for example.

Working with Rulers and Paragraph Styles

Text objects also provide for a ruler, by which the user can edit paragraph attributes such as indents and tabs. NSTextView works with the NSRulerView class and uses the NSTextStorage and NSParagraphStyle classes to handle paragraph attributes. To show or hide a text object’s ruler, use the **toggleRuler:** action method. Similar to the Font Panel, NSTextView can be set not to use a ruler with the **setUsesRuler:** method.

Adopted Protocols

NSChangeSpelling

– changeSpelling:

NSIgnoreMisspelledWords

– ignoreSpelling:

Method Types

Creating instances

– initWithFrame:

Getting the characters

– string

Setting graphic attributes

– setBackgroundColor:
– backgroundColor
– setDrawsBackground:
– drawsBackground

Setting behavioral attributes

– setEditable:
– isEditable
– setSelectable:
– isSelectable
– setFieldEditor:
– isFieldEditor
– setRichText:
– isRichText
– setImportsGraphics:
– importsGraphics

Using the Font Panel and menu

– setUsesFontPanel:
– usesFontPanel

Using the ruler

– toggleRuler:
– isRulerVisible

Changing the selection

– setSelectedRange:
– selectedRange

Replacing text

– replaceCharactersInRange:withRTF:
– replaceCharactersInRange:withRTFD:
– replaceCharactersInRange:withString:
– setString:

Action methods for editing

– selectAll:
– copy:
– cut:
– paste:
– copyFont:
– pasteFont:
– copyRuler:
– pasteRuler:
– delete:

Changing the font

– changeFont:
– setFont:
– setFont:range:
– font

Setting text alignment

– setAlignment:
– alignCenter:
– alignLeft:
– alignRight:
– alignment

Setting text color

– setTextColor:
– setTextColor:range:
– textColor

Setting super- and subscripting

– superscript:
– subscript:
– unscript:

Underlining text

– underline:

Reading and writing RTF

– readRTFDFromFile:
– writeRTFDToFile:atomically:
– RTFDFromRange:
– RTFFromRange:

Checking spelling

– checkSpelling:
– showGuessPanel:

Constraining size

– setMaxSize:
– maxSize
– setMinSize:
– minSize
– setVerticallyResizable:
– isVerticallyResizable
– setHorizontallyResizable:
– isHorizontallyResizable
– sizeToFit

Scrolling

– scrollRangeToVisible:

Setting the delegate

– setDelegate:
– delegate

Instance Methods

alignCenter:

– (void)**alignCenter:**(id)*sender*

This action method applies center alignment to selected paragraphs (or all text if the receiver is a plain text object).

See also: – **alignLeft:**, – **alignRight:**, – **alignment**, – **setAlignment:**

alignLeft:

– (void)**alignLeft:**(id)*sender*

This action method applies left alignment to selected paragraphs (or all text if the receiver is a plain text object).

See also: – **alignCenter:**, – **alignRight:**, – **alignment**, – **setAlignment:**

alignRight:

– (void)**alignRight:**(id)*sender*

This action method applies right alignment to selected paragraphs (or all text if the receiver is a plain text object).

See also: – **alignLeft:**, – **alignCenter:**, – **alignment**, – **setAlignment:**

alignment

– (NSTextAlignment)**alignment**

Returns the alignment of the first selected paragraph, or of all text for a plain text object. This value is one of:

NSLeftTextAlignment
NSRightTextAlignment

NSCenterTextAlignment
NSJustifiedTextAlignment
NSNaturalTextAlignment (realized as one of the above depending on the script)

backgroundColor

– (NSColor *)**backgroundColor**

Returns the receiver's background color.

See also: – **drawsBackground**, – **setBackgroundColor:**

changeFont:

– (void)**changeFont:**(id)*sender*

This action method changes the font of the selection for a rich text object, or of all text for a plain text object. If the receiver doesn't use the Font Panel, however, this method does nothing.

This method changes the font by sending **convertFont:** messages to *sender* (which is presumed to be an NSFontManager or similarly capable object) and applying each NSFont returned to the appropriate text. If a rich text object's selection contains multiple fonts, **convertFont:** is invoked for each contiguous range of characters that share a font. See the NSFontManager class specification for more information on font conversion.

See also: – **usesFontPanel**

checkSpelling:

– (void)**checkSpelling:**(id)*sender*

This action method searches for a misspelled word in the receiver's text. The search starts at the end of the selection and continues until it reaches a word suspected of being misspelled or the end of the text. If a word isn't recognized by the spelling server. A **showGuessPanel:** message then opens the Guess panel and allows the user to make a correction or add the word to the local dictionary.

See also: – **showGuessPanel:**

copy:

– (void)**copy:**(id)*sender*

This action method copies the selected text onto the general pasteboard, in as many formats as the receiver supports. A plain text object uses NSStringPboardType for plain text, and a rich text object also uses NSRTFPboardType.

See also: – **copyFont:**, – **copyRuler:**, – **cut:**, – **paste:**

copyFont:

– (void)**copyFont:**(id)*sender*

This action method copies the font information for the first character of the selection (or for the insertion point) onto the font pasteboard, as NSFontPboardType.

See also: – **copy:**, – **copyRuler:**, – **cut:**, – **paste:**

copyRuler:

– (void)**copyRuler:**(id)*sender*

This action method copies the paragraph style information for first selected paragraph onto the ruler pasteboard, as NSRulerPboardType, and expands the selection to paragraph boundaries.

See also: – **copy:**, – **copyFont:**, – **cut:**, – **paste:**

cut:

– (void)**cut:**(id)*sender*

This action method deletes the selected text and places it onto the general pasteboard, in as many formats as the receiver supports. A plain text object uses NSStringPboardType for plain text, and a rich text object also uses NSRTFPboardType.

See also: – **delete:**, – **copy:**, – **copyFont:**, – **copyRuler:**, – **paste:**

delegate

– (id)**delegate**

Returns the receiver's delegate, or **nil** if it has none.

See also: – **setDelegate:**

delete:

– (void)**delete:**(id)*sender*

This action method deletes the selected text.

See also: – **cut:**

drawsBackground

– (BOOL)**drawsBackground**

Returns YES if the receiver draws its background, NO if it doesn't.

See also: – **backgroundColor**, – **setDrawsBackground:**

font

– (NSFont *)**font**

Returns the font of the first character in the receiver's text, or of the insertion point if there's no text.

See also: – **setFont:**, – **setFont:range:**

importsGraphics

– (BOOL)**importsGraphics**

Returns YES if the receiver allows the user to import files by dragging, NO if it doesn't. A text object that accepts dragged files is also a rich text object.

See also: – **isRichText**, – **setImportsGraphics:**

initWithFrame:

– (id)**initWithFrame:**(NSRect)*frameRect*

Initializes the receiver with *frameRect* as its frame rectangle. This method actually substitutes an instance of a concrete subclass of NSText, such as NSTextView, depending on the platform, and configures that instance to archive itself in a manner portable across OpenStep implementations.

isEditable

– (BOOL)**isEditable**

Returns YES if the receiver allows the user to edit text, NO if it doesn't. You can change the receiver's text programmatically regardless of this setting.

If the receiver is editable, it's also selectable.

See also: – **isSelectable**, – **setEditable:**

isFieldEditor

– (BOOL)**isFieldEditor**

Returns YES if the receiver interprets Tab, Shift-Tab, and Return (Enter) as cues to end editing, and possibly to change the first responder; no if it accepts them as text input. See the NSWindow class specification for more information on field editors. By default, NSText objects don't behave as field editors.

See also: – **setFieldEditor:**

isHorizontallyResizable

– (BOOL)**isHorizontallyResizable**

Returns YES if the receiver automatically changes its width to accommodate the width of its text, NO if it doesn't.

See also: – **isVerticallyResizable**, – **setHorizontallyResizable:**

isRichText

– (BOOL)**isRichText**

Returns YES if the receiver allows the user to apply attributes to specific ranges of the text, NO if it doesn't.

See also: – **importsGraphics**, – **setRichText:**

isRulerVisible

– (BOOL)**isRulerVisible**

Returns YES if the receiver's enclosing scroll view shows its ruler, NO otherwise.

See also: – **toggleRuler:**

isSelectable

– (BOOL)**isSelectable**

Returns YES if the receiver allows the user to select text, NO if it doesn't.

See also: – **isEditable**, – **setSelectable:**

isVerticallyResizable

– (BOOL)**isVerticallyResizable**

Returns YES if the receiver automatically changes its height to accommodate the height of its text, NO if it doesn't.

See also: – **isHorizontallyResizable**, – **setVerticallyResizable:**

maxSize

– (NSSize)**maxSize**

Returns the receiver's maximum size.

See also: – **minSize**, – **setMaxSize:**

minSize

– (NSSize)**minSize**

Returns the receiver's minimum size.

See also: – **maxSize**, – **setMinSize:**

paste:

– (void)**paste:**(id)*sender*

This action method pastes text from the general pasteboard at the insertion point or over the selection.

See also: – **copy:**, – **cut:**, – **pasteFont:**, – **pasteRuler:**

pasteFont:

– (void)**pasteFont:**(id)*sender*

This action method pastes font information from the font pasteboard onto the selected text or insertion point of a rich text object, or over all text of a plain text object.

See also: – **copyFont:**, – **pasteRuler:**

pasteRuler:

– (void)**pasteRuler:**(id)*sender*

This action method pastes paragraph style information from the ruler pasteboard onto the selected paragraphs of a rich text object. It doesn't apply to a plain text object.

See also: – **copyFont:**, – **pasteRuler:**

readRTFDFromFile:

– (BOOL)**readRTFDFromFile:**(NSString *)*path*

Attempts to read the RTFD file at *path*, returning YES if successful and NO if not. *path* should be the path for an **.rtf** file or an **.rtfd** file wrapper, not for the RTF file within an **.rtfd** file wrapper. *<<Or not even a **.rtf** file?>>*

See also: – **writeRTFDToFile:atomically:**

replaceCharactersInRange:withRTF:

– (void)**replaceCharactersInRange:**(NSRange)*aRange* **withRTF:**(NSData *)*rtfData*

Replaces the characters in *aRange* with RTF text interpreted from *rtfData*. This method applies only to rich text objects.

See also: – **replaceCharactersInRange:withRTFD:**, – **replaceCharactersInRange:withString:**

replaceCharactersInRange:withRTFD:

– (void)**replaceCharactersInRange:**(NSRange)*aRange* **withRTFD:**(NSData *)*rtfdData*

Replaces the characters in *aRange* with RTFD text interpreted from *rtfdData*. This method applies only to rich text objects.

See also: – **replaceCharactersInRange:withRTF:**, – **replaceCharactersInRange:withString:**

replaceCharactersInRange:withString:

– (void)**replaceCharactersInRange:**(NSRange)*aRange* **withString:**(NSString *)*aString*

Replaces the characters in *aRange* with *aString*. For a rich text object, the text of *aString* is assigned the formatting attributes of the first character of the text it replaces, or of the character immediately before *aRange* if the range's length is zero. If the range's location is zero, the formatting attributes of the first character in the receiver are used.

See also: – **replaceCharactersInRange:withRTF:**, – **replaceCharactersInRange:withRTFD:**

RTFDFromRange:

– (NSData *)**RTFDFromRange:**(NSRange)*aRange*

Returns an NSData object that contains an RTFD stream corresponding to the characters and attributes within *aRange*. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

When writing data to the pasteboard, you can use the NSData object as the first argument to NSPasteboard's **setData:forType:** method, with a second argument of NSRTFDPboardType.

See also: – **RTFFromRange:**

RTFFromRange:

– (NSData *)**RTFFromRange:**(NSRange)*range*

Returns an NSData object that contains an RTF stream corresponding to the characters and attributes within *aRange*, omitting any attachment characters and attributes. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

When writing data to the pasteboard, you can use the NSData object as the first argument to NSPasteboard's **setData:forType:** method, with a second argument of NSRTFPboardType.

See also: – **RTFDFromRange:**

scrollRangeToVisible:

– (void)**scrollRangeToVisible:**(NSRange)*aRange*

Scrolls the receiver in its enclosing scroll view so that the first characters of *aRange* are visible.

selectAll:

– (void)**selectAll:**(id)*sender*

This action method selects all of the receiver's text.

selectedRange

– (NSRange)**selectedRange**

Returns the range of selected characters.

See also: – **setSelectedRange:**

setAlignment:

– (void)**setAlignment:**(NSTextAlignment)*mode*

Sets the alignment of all the receiver's text to *mode*, which may be one of:

NSLeftTextAlignment
NSRightTextAlignment
NSCenterTextAlignment
NSJustifiedTextAlignment
NSNaturalTextAlignment (realized as one of the above depending on the script)

See also: – **alignment**, – **alignLeft:**, – **alignCenter:**, – **alignRight:**

setBackgroundColor:

– (void)**setBackgroundColor:**(NSColor *)*aColor*

Sets the receiver's background color to *aColor*.

See also: – **setDrawsBackground:**, – **backgroundColor**

setDelegate:

– (void)**setDelegate:**(id)*anObject*

Sets the receiver's delegate to *anObject*, without retaining it.

See also: – **delegate**

setDrawsBackground:

– (void)**setDrawsBackground:**(BOOL)*flag*

Controls whether the receiver draws its background. If *flag* is YES, the receiver fills its background with the background color; if *flag* is NO, it doesn't.

See also: – **setBackgroundColor:**, – **drawsBackground**

setEditable:

– (void)**setEditable:**(BOOL)*flag*

Controls whether the receiver allows the user to edit its text. If *flag* is YES, the receiver allows the user to edit text and attributes; if *flag* is NO, it doesn't. You can change the receiver's text programmatically regardless of this setting. If the receiver is made editable, it's also made selectable. NSText objects are by default editable.

See also: – **setSelectable:**, – **isEditable**

setFieldEditor:

– (void)**setFieldEditor:**(BOOL)*flag*

Controls whether the receiver interprets Tab, Shift-Tab, and Return (Enter) as cues to end editing, and possibly to change the first responder. If *flag* is YES, it does; if *flag* is NO, it doesn't, instead accepting these characters as text input. See the NSWindow class specification for more information on field editors. By default, NSText objects don't behave as field editors.

See also: – **isFieldEditor**

setFont:

– (void)**setFont:**(NSFont *)*aFont*

Sets the font of all the receiver's text to *aFont*.

See also: – **setFont:range:**, – **font**

setFont:range:

– (void)**setFont:**(NSFont *)*aFont* **range:**(NSRange)*aRange*

Sets the font of characters within *aRange* to *aFont*. This method applies only to a rich text object.

See also: – **setFont:**, – **font**

setHorizontallyResizable:

– (void)**setHorizontallyResizable:**(BOOL)*flag*

Controls whether the receiver changes its width to fit the width of its text. If *flag* is YES it does; if *flag* is NO it doesn't.

See also: – **setVerticallyResizable:**, – **isHorizontallyResizable**

setImportsGraphics:

– (void)**setImportsGraphics:**(BOOL)*flag*

Controls whether the receiver allows the user to import files by dragging. If *flag* is YES, it does; if *flag* is NO, it doesn't. If the receiver is set to accept dragged files, it's also made a rich text object. Subclasses may or may not accept dragged files by default.

See also: – **setRichText:**, – **importsGraphics**

setMaxSize:

– (void)**setMaxSize:**(NSSize)*aSize*

Sets the receiver's maximum size to *aSize*.

See also: – **setMinSize:**, – **maxSize**

setMinSize:

– (void)**setMinSize:**(NSSize)*aSize*

Sets the receiver's minimum size to *aSize*.

See also: – **setMaxSize:**, – **minSize**

setRichText:

– (void)**setRichText:**(BOOL)*flag*

Controls whether the receiver allows the user to apply attributes to specific ranges of the text. If *flag* is YES it does; if *flag* is NO it doesn't. If *flag* is NO, the receiver is also set not to accept dragged files. Subclasses may or may not let the user apply multiple attributes to the text and accept drag files by default.

See also: – **isRichText**, – **setImportsGraphics:**

setSelectable:

– (void)**setSelectable:**(BOOL)*flag*

Controls whether the receiver allows the user to select its text. If *flag* is YES, the receiver allows the user to select text; if *flag* is NO, it doesn't. You can set selections programmatically regardless of this setting. If the receiver is made not selectable, it's also made not editable. NSText objects are by default editable and selectable.

See also: – **setEditable:**, – **isSelectable**

setSelectedRange:

– (void)**setSelectedRange:**(NSRange)*aRange*

Selects the receiver's characters within *aRange*.

See also: – **selectedRange**

setString:

– (void)**setString:**(NSString *)*aString*

Replaces the receiver's entire text with *aString*, applying the formatting attributes of the old first character to its new contents.

setTextColor:

– (void)**setTextColor:**(NSColor *)*color*

Sets the text color of all characters in the receiver to *aColor*. Removes the text color attribute if *aColor* is **nil**.

See also: – **setTextColor:range:**, – **textColor**

setTextColor:range:

– (void)**setTextColor:**(NSColor *)*aColor* **range:**(NSRange)*aRange*

Sets the text color of characters within *aRange* to *aColor*. Removes the text color attribute if *aColor* is **nil**. This method applies only to rich text objects.

See also: – **setTextColor:**, – **textColor**

setUsesFontPanel:

– (void)**setUsesFontPanel:**(BOOL)*flag*

Controls whether the receiver uses the Font Panel and Font menu. If *flag* is YES, the receiver responds to messages from the Font Panel and from the Font menu, and updates the Font Panel with the selection font whenever it changes. If *flag* is NO the receiver doesn't do any of this. By default, an NSText object uses the Font Panel and menu.

See also: – **usesFontPanel**

setVerticallyResizable:

– (void)**setVerticallyResizable:**(BOOL)*flag*

Controls whether the receiver changes its height to fit the height of its text. If *flag* is YES it does; if *flag* is NO it doesn't.

See also: – **setHorizontallyResizable:**, – **isVerticallyResizable**

showGuessPanel:

– (void)**showGuessPanel:**(id)*sender*

This action method opens the Spelling panel, allowing the user to make a correction during spell checking.

See also: – **checkSpelling:**

sizeToFit

– (void)**sizeToFit**

Resizes the receiver to fit its text.

See also: – **isHorizontallyResizable**, – **isVerticallyResizable**

string

– (NSString *)**string**

Returns the characters of the receiver's text.

See also: – **setString:**

subscript:

– (void)**subscript:**(id)*sender*

This action method applies a subscript attribute to selected text (or all text if the receiver is a plain text object), lowering its baseline offset by a predefined amount.

See also: – **subscript:**, – **unscript:**, – **lowerBaseline:** (NSTextView)

superscript:

– (void)**superscript:**(id)*sender*

This action method applies a superscript attribute to selected text (or all text if the receiver is a plain text object), raising its baseline offset by a predefined amount.

See also: – **subscript:**, – **unscript:**, – **raiseBaseline:** (NSTextView)

textColor

– (NSColor *)**textColor**

Returns the color of the receiver's first character, or for the insertion point if there's no text.

See also: – **setTextColor:**, – **setTextColor:range:**

toggleRuler:

– (void)**toggleRuler:**(id)*sender*

This action method shows or hides the ruler, if the receiver is enclosed in a scroll view.

underline:

– (void)**underline:**(id)*sender*

This action method underlines selected text for a rich text object, or all text for a plain text object.

unscript:

– (void)**unscript:**(id)*sender*

This action method removes any superscripting or subscripting from selected text (or all text if the receiver is a plain text object).

See also: – **subscript:**, – **superscript:**, – **raiseBaseline:** (NSTextView), – **lowerBaseline:** (NSTextView)

usesFontPanel

– (BOOL)**usesFontPanel**

Returns YES if the receiver uses the Font Panel, NO otherwise.

See also: – **setUsesFontPanel:**

writeRTFDToFile:atomically:

– (BOOL)**writeRTFDToFile:**(NSString *)*path* **atomically:**(BOOL)*flag*

Writes the receiver's text as RTF with attachments to a file or directory at *path*. Returns YES on success and NO on failure. If *atomicFlag* is YES, attempts to write the file safely so that an existing file at *path* is not overwritten, nor does a new file at *path* actually get created, unless the write is successful.

See also: – **RTFFromRange:**, – **RTFDFromRange:**, – **readRTFDFromFile:**

Methods Implemented By the Delegate

textDidBeginEditing:

– (void)**textDidBeginEditing:**(NSNotification *)*aNotification*

Informs the delegate that the text object has begun editing (that it has become first responder). The name of *aNotification* is NSTextViewDidBeingEditingNotification.

textDidChange:

– (void)**textDidChange:**(NSNotification *)*aNotification*

Informs the delegate that the text object has changed its characters or formatting attributes. The name of *aNotification* is NSTextViewDidChangeNotification.

textDidEndEditing:

– (void)**textDidEndEditing:**(NSNotification *)*aNotification*

Informs the delegate that the text object has finished editing (that it has resigned first responder status). The name of *aNotification* is NSTextViewDidEndEditingNotification.

textShouldBeginEditing:

– (BOOL)**textShouldBeginEditing:**(NSText *)*aTextObject*

Invoked from a text object's implementation of **becomeFirstResponder**, this method requests permission for *aTextObject* to begin editing. If the delegate returns YES, the text object proceeds to make changes. If the delegate returns NO, the text object abandons the editing operation. This method is invoked whenever *aTextObject* attempts to become first responder.

See also: – **makeFirstResponder:** (NSWindow), – **becomeFirstResponder** (NSResponder)

textShouldEndEditing:

– (BOOL)**textShouldEndEditing:**(NSText *)*aTextObject*

Invoked from a text object's implementation of **resignFirstResponder**, this method requests permission for *aTextObject* to end editing. If the delegate returns YES, the text object proceeds to finish editing and resign first responder status. If the delegate returns NO, the text object selects all of its text and remains the first responder.

See also: – **resignFirstResponder** (NSResponder)

Notifications

NSTextDidBeginEditingNotification

Posted when an NSText object begins any operation that changes characters or formatting attributes.

The notification contains a notification object but no userInfo dictionary. The notification object is the notifying NSText object.

NSTextDidChangeNotification

Posted after an NSText object performs any operation that changes characters or formatting attributes.

The notification contains a notification object but no userInfo dictionary. The notification object is the notifying NSText object.

NSTextDidEndEditingNotification

<< Forthcoming. >> NeXTSTEP

NSTextAttachment

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: AppKit/NSTextAttachment.h

Class Description

NSTextAttachment objects are used by the NSAttributedString class cluster as the values for attachment attributes (stored in the attributed string under the key named NSAttachmentAttributeName). The objects you create with this class are referred to as *text attachment objects*, or when no confusion will result, as *text attachments* or merely *attachments*. See the NSAttributedString and NSTextView class specifications for general information on text attachments.

A text attachment object contains an NSFileWrapper, which in turn holds the contents of the attached file. It also uses a cell object conforming to the NSTextAttachmentCell protocol to draw and handle mouse events. Most of the behavior of a text attachment is relegated to the file wrapper and the attachment cell. See the corresponding class and protocol specifications for more information.

Method Types

Creating an NSTextAttachment

– initWithFileWrapper:

Setting the file wrapper

– setFileWrapper:
– fileWrapper

Setting the attachment cell

– setAttachmentCell:
– attachmentCell

Instance Methods

attachmentCell

– (id <NSTextAttachmentCell>)**attachmentCell**

Returns the object used to draw the icon for the attachment and to handle mouse events. An NSTextAttachment by default uses an NSTextAttachmentCell that displays the attached file's icon, or its contents if the file contains an image.

See also: – **fileWrapper**, – **image** (NSCell), – **icon** (NSFileWrapper), – **setAttachmentCell:**

fileWrapper

– (NSFileWrapper *)**fileWrapper**

Returns the receiver's file wrapper, which holds the contents of the attached file.

See also: – **setFileWrapper:**

initWithFileWrapper:

– (id)**initWithFileWrapper:**(NSFileWrapper *)*aWrapper*

Initializes a newly allocated NSTextAttachment to contain *aWrapper* and to use an NSTextAttachmentCell as its attachment cell. If *aWrapper* contains an image file that the receiver can interpret as an NSImage object, it sets the attachment cell's image to the NSImage rather than to *aWrapper*'s icon.

This method is the designated initializer for the NSTextAttachment class. Returns **self**.

See also: – **setFileWrapper:**, – **setAttachmentCell:**

setAttachmentCell:

– (void)**setAttachmentCell:**(id <NSTextAttachmentCell>)*aCell*

Sets the object used to draw the icon for the attachment and to handle mouse events to *aCell*.

See also: – **setFileWrapper:**, – **setImage:** (NSCell), – **icon** (NSFileWrapper), – **attachmentCell**

setFileWrapper:

– (void)**setFileWrapper:**(NSFileWrapper *)*aWrapper*

Sets the receiver's file wrapper, which holds the contents of the attached file, to *aWrapper*.

See also: – **fileWrapper**

NSTextAttachmentCell

Inherits From: NSCell : NSObject

Conforms To: NSTextAttachmentCell
NSObject (NSObject)

Declared In: NSTextAttachment.h

Class Description

NSTextAttachmentCell implements the functionality of the NSTextAttachmentCell protocol. See the NSTextAttachment protocol specification for a general discussion of the protocol's methods. This specification describes only those methods whose implementations have features peculiar to this class.

See the NSAttributedString and NSTextView class specifications for general information on text attachments.

Adopted Protocols

NSTextAttachmentCell

– attachment
– cellBaselineOffset
– cellSize
– drawWithFrame:inView:
– highlight:withFrame:inView:
– trackMouse:inRect:ofView:untilMouseUp:
– setAttachment:
– wantsToTrackMouse

Instance Methods

trackMouse:inRect:ofView:untilMouseUp:

@protocol NSTextAttachmentCell
– (BOOL)**trackMouse:**(NSEvent *)*theEvent*
inRect:(NSRect)*cellFrame*
ofView:(NSView *)*aTextView*
untilMouseUp:(BOOL)*flag*

Handles a mouse-down event on the receiver's image. NSTextAttachmentCell's implementation of this method calls upon *aTextView*'s delegate to handle the event. If *theEvent* concludes as a double click, the text attachment cell sends the delegate a **textView:doubleClickedOnCell:inRect:** message and returns YES. Otherwise, depending on whether the user clicks or drags the cell, it sends the delegate a **textView: clickedOnCell:inRect:** or a **textView:draggingCell:inRect:event:** message and returns YES. NSTextAttachmentCell's implementation returns NO only if *flag* is NO and the mouse is dragged outside of *cellFrame*. The delegate methods are invoked only if the delegate can respond to them.

See also: – **wantsToTrackMouse**, – **trackMouse:inRect:ofView:untilMouseUp:** (NSCell), – **lockFocus** (NSView)

wantsToTrackMouse

@protocol NSTextAttachmentCell
– (BOOL)**wantsToTrackMouse**

Returns YES. NSTextAttachmentCell objects support dragging. An NSTextView invokes this method before sending **trackMouse:inRect:ofView:untilMouseUp:** to the text attachment cell.

A more static subclass might override this method to return NO, which results in the attachment image behaving as any other glyph in the text, and not allow itself to be dragged or to perform a method on being clicked.

NSTextContainer

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: AppKit/NSTextContainer.h

Class Description

An NSTextContainer defines a region where text is laid out. An NSLayoutManager uses NSTextContainers to determine where to break lines, lay out portions of text, and so on. NSTextContainer defines rectangular regions, but you can create subclasses that define regions of other shapes, such as circular regions, regions with holes in them, or regions that flow alongside graphics.

You normally use an NSTextView to display the text laid out within an NSTextContainer. An NSTextView can have only one NSTextContainer; however, since the two are separate objects, you can replace an NSTextView's container to change the layout of the text it displays. You can also display an NSTextContainer's text in any NSView by locking the graphics focus on it and using NSLayoutManager's **drawBackgroundForGlyphRange:atPoint:** and **drawGlyphsForGlyphRange:atPoint:** methods. If you have no need of actually displaying the text—if you're only calculating line breaks or number of lines or pages, for example—you can use an NSTextContainer without an NSTextView.

Region, Bounding Rectangle, and Inset

An NSTextContainer's region is defined within a *bounding rectangle* whose coordinate system starts at (0, 0) in the top left corner. The size of this rectangle is returned by the **containerSize** method and set using **setContainerSize:**. You can define a container's region so that it's always the same shape, such as a circle whose diameter is the narrower of the bounding rectangle's dimensions, or you can define the region relative to the bounding rectangle, such as an oval region that fits inside the bounding rectangle (and that's a circle when the bounding rectangle is square). Regardless of a text container's shape, its NSTextView always clips drawing to its bounding rectangle.

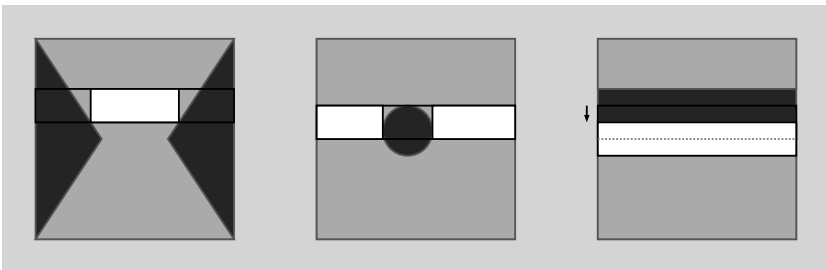
A subclass of NSTextContainer defines its region by overriding three methods. The first, **isSimpleRectangularTextContainer**, indicates whether the region is currently a nonrotated rectangle, thus allowing the NSLayoutManager to optimize layout of text (since custom NSTextContainers typically define more complex regions, your implementation of this method will probably return NO). The second method, **containsPoint:**, is used for testing mouse events and determines whether or not a given point lies in the region. The third method is used for the actual layout of text, defining the region in terms of rectangles available to lay text in; this process is described in "Calculating Text Layout".

An NSTextContainer usually covers its NSTextView exactly, but can be inset within the view frame with NSTextView's **setTextContainerInset:** method. The NSTextContainer's bounding rectangle from the inset position then establishes the limits of the NSTextContainer's region. The inset also helps to determine the size of the bounding rectangle when the NSTextContainer tracks the height or width of its NSTextView, as described in "Tracking the Size of the NSTextView".

Calculating Text Layout

An NSLayoutManager lays text within an NSTextContainer in lines of glyphs, running either horizontally or vertically. The layout of these lines within an NSTextContainer is determined by its shape. For example, if the NSTextContainer is narrower in some parts than in others, the lines in those parts must be shortened; if there are holes in the region, some lines must be fragmented; if there's a gap across the entire region, the lines that would overlap it have to be shifted to compensate. This is illustrated in the figure below.

Note: The text system currently supports only horizontal text layout.



The NSLayoutManager proposes a rectangle for a given line, and then asks the NSTextContainer to adjust the rectangle to fit. The proposed rectangle usually spans the NSTextContainer's bounding rectangle, but it can be narrower or wider, and it can also lie partially or completely outside the bounding rectangle. The message that an NSLayoutManager sends the container to adjust the proposed rectangle is **lineFragmentRectForProposedRect:sweepDirection:movementDirection:remainingRect:**, which returns the largest rectangle available for the proposed rectangle, based on the direction text is laid out. It also returns a rectangle containing any remaining space, such as that left on the other side of a hole or gap in the NSTextContainer.

Text is laid out along lines that run either horizontally or vertically, and in either direction. This type of movement is called the *sweep direction* and is expressed by the NSLineSweepDirection type. The direction in which the lines progress is then called the *line movement direction* and is expressed by the NSLineMovementDirection type. Each affects the adjustment of a line fragment rectangle in a different

way: The rectangle can be moved or shortened along the sweep direction and shifted (but not resized) in the line movement direction.

NSLineSweepDirection values	NSLineMovementDirection values
NSLineSweepLeft	NSLineMovesLeft
NSLineSweepRight	NSLineMovesRight
NSLineSweepDown	NSLineMovesDown
NSLineSweepUp	NSLineMovesUp
	NSLineDoesntMove

For the three examples above, the sweep direction is NSLineSweepRight and the line movement direction is NSLineMovesDown. In the first example, the proposed rectangle spans the region's bounding rectangle and is shortened by the text container to fit inside the hourglass shape with no remainder.

In the second example, the proposed rectangle crosses a hole, so the text container must return a shorter rectangle (the white rectangle on the left) along with a remainder (the white rectangle on the right). The next rectangle proposed by the NSLayoutManager will then be this remainder rectangle, and will be returned unchanged by the text container.

In the third example, a gap crosses the entire NSTextContainer. Here the text container shifts the proposed rectangle down until it lies completely within the container's region. If the line movement direction here were NSLineDoesntMove, the NSTextContainer would have to return NSZeroRect indicating that the line simply doesn't fit. In such a case it's up to the NSLayoutManager to propose a different rectangle or to move on to a different container. When a text container shifts a line fragment rectangle, the layout manager takes this into account for subsequent lines.

The NSLayoutManager makes one final adjustment when it actually fits text into the rectangle. This adjustment is a small amount fixed by the NSTextContainer, called the *line fragment padding*, which defines the portion on each end of the line fragment rectangle left blank. Text is inset within the line fragment rectangle by this amount (the rectangle itself is unaffected). Padding allows for small-scale adjustment of the NSTextContainer's region at the edges and around any holes, and keeps text from abutting any other graphics displayed near the region. You can change the padding from its default value with the **setLineFragmentPadding:** method, or override the default in your subclass. Note that line fragment padding isn't a suitable means for expressing margins; you should set the NSTextView's position and size for document margins or the paragraph margin attributes for text margins.

Tracking the Size of the NSTextView

Normally, if you resize an NSTextView its NSTextContainer doesn't change in size. You can, however, set an NSTextContainer to track the size of its NSTextView and adjust its own size to match whenever the NSTextView's size changes. The **setHeightTracksTextView:** and **setWidthTracksTextView:** methods allow you to control this tracking for either dimension.

When an NSTextContainer adjusts its size to match that of its NSTextView, it takes into account the inset specified by the NSTextView so that the bounding rectangle is inset from every edge possible. In other words, an NSTextContainer that tracks the size of its NSTextView is always smaller than the NSTextView (in the appropriate dimension) by twice the inset. Suppose an NSTextContainer is set to track width and its NSTextView gives it an inset of (10, 10). Now, if the NSTextView's width is changed to 138, the NSTextContainer's top left corner is set to lie at (10, 10) and its width is set to 118, so that its right edge is 10 points from the NSTextView's right edge. Its height remains the same.

Whether it tracks the size of its NSTextView or not, an NSTextContainer doesn't grow or shrink as text is added or deleted; instead, the NSLayoutManager resizes the NSTextView based on the portion of the NSTextContainer actually filled with text. To allow an NSTextView to be resized in this manner, use NSTextView's **setVerticallyResizable:** or **setHorizontallyResizable:** methods as needed, set the text container not to track the size of its text view, and set the text container's size in the appropriate dimension large enough to accommodate a great amount of text—about 1e7 (this incurs no cost whatever in processing or storage).

Note that an NSTextView can be resized based on its NSTextContainer, and an NSTextContainer can resize itself based on its NSTextView. If you set both objects up to resize automatically in the same dimension, your application can get trapped in an infinite loop. When text is added to the NSTextContainer, the NSTextView is resized to fit the area actually used for text; this causes the NSTextContainer to resize itself and relay its text, which causes the NSTextView to resize itself again, and so on ad infinitum. Each type of size tracking has its proper uses; be sure to use only one for either dimension.

Method Types

Creating an instance

– initWithContainerSize:

Managing text components

– setLayoutManager:
– layoutManager
– replaceLayoutManager:
– setTextView:
– textView

Controlling size

– setContainerSize:
– containerSize
– setWidthTracksTextView:
– widthTracksTextView
– setHeightTracksTextView:
– heightTracksTextView

Setting line fragment padding

– setLineFragmentPadding:
– lineFragmentPadding

Calculating text layout

– lineFragmentRectForProposedRect:sweepDirection: movementDirection:remainingRect:
– isSimpleRectangularTextContainer

Mouse hit testing

– containsPoint:

Instance Methods

containerSize

– (NSSize)**containerSize**

Returns the size of the receiver's bounding rectangle, regardless of the size of its region.

See also: – **textContainerInset** (NSTextView), – **setContainerSize:**

containsPoint:

– (BOOL)**containsPoint:**(NSPoint)*aPoint*

Overridden by subclasses to returns YES if *aPoint* lies within the receiver's region or on the region's edge—not simply within its bounding rectangle—NO otherwise. For example, if the receiver defines a donut shape and *aPoint* lies in the hole, this method returns NO. This method can be used for hit testing of mouse events.

NSTextContainer's implementation merely checks that *aPoint* lies within its bounding rectangle.

heightTracksTextView

– (BOOL)**heightTracksTextView**

Returns YES if the receiver adjusts the height of its bounding rectangle when its NSTextView is resized, NO otherwise. The height is adjusted to the height of the NSTextView minus twice the inset height (as given by NSTextView's **textContainerInset** method).

See the class description for more information on size tracking.

See also: – **widthTracksTextView**, – **setHeightTracksTextView:**

initWithContainerSize:

– (id)**initWithContainerSize:**(NSSize)*aSize*

Initializes the receiver, a newly allocated NSTextContainer, with *aSize* as the size of its bounding rectangle. The new NSTextContainer must be added to an NSLayoutManager before it can be used; it must also have an NSTextView set for text to be displayed. This method is the designated initializer for the NSTextContainer class. Returns **self**.

See also: – **addTextContainer:** (NSLayoutManager), – **setTextView:**

isSimpleRectangularTextContainer

– (BOOL)**isSimpleRectangularTextContainer**

Overridden by subclasses to return YES if the receiver's region is a rectangle with no holes or gaps and whose edges are parallel to the NSTextView's coordinate system axes; returns NO otherwise. An NSTextContainer whose shape changes can return YES if its region is currently a simple rectangle, but when its shape does change it must send **textContainerChangedGeometry:** to its NSLayoutManager so the layout can be recalculated.

NSTextContainer's implementation of this method returns YES.

layoutManager

– (NSLayoutManager *)**layoutManager**

Returns the receiver's NSLayoutManager.

See also: – **setLayoutManager:**, – **replaceLayoutManager:**

lineFragmentPadding

– (float)**lineFragmentPadding**

Returns the amount (in points) by which text is inset within line fragment rectangles.

See also: – **lineFragmentRectForProposedRect:sweepDirection:movementDirection: remainingRect:**, – **setLineFragmentPadding:**

lineFragmentRectForProposedRect:sweepDirection:movementDirection: remainingRect:

– (NSRect)**lineFragmentRectForProposedRect:**(NSRect)*proposedRect*
sweepDirection:(NSLineSweepDirection)*sweepDirection*
movementDirection:(NSLineMovementDirection)*movementDirection*
remainingRect:(NSRect *)*remainingRect*

Overridden by subclasses to calculate and return the longest rectangle available for *proposedRect* for displaying text, or NSZeroRect if there is none according to the receiver's region definition.

There is no guarantee as to the width of the proposed rectangle or to its location. For example, the proposed rectangle is likely to be much wider than the width of the receiver. The receiver should examine *proposedRect* to see that it intersects its bounding rectangle, and should return a modified rectangle based on *sweepDirection* and *movementDirection*, whose possible values are listed in the class description. If *sweepDirection* is NSLineSweepRight, for example, the receiver uses this information to trim the right end of *proposedRect* as needed rather than the left end.

If *proposedRect* doesn't completely overlap the region along the axis of *movementDirection* and *movementDirection* isn't NSLineDoesntMove, this method can either shift the rectangle in that direction as much as needed so that it does completely overlap, or return NSZeroRect to indicate that the proposed rectangle simply doesn't fit.

Upon returning, *remainingRect* contains the unused, possibly shifted, portion of *proposedRect* that's available for further text, or NSZeroRect if there is no remainder.

See the class description for more information on overriding this method.

replaceLayoutManager:

– (void)**replaceLayoutManager:**(NSLayoutManager *)*aLayoutManager*

Replaces the NSLayoutManager for the group of text-system objects containing the receiver with *aLayoutManager*. All NSTextContainers and NSTextViews sharing the original NSLayoutManager then share the new one. This method makes all the adjustments necessary to keep these relationships intact, unlike **setLayoutManager:**.

See also: – **layoutManager**

setContainerSize:

– (void)**setContainerSize:**(NSSize)*aSize*

Sets the size of the receiver's bounding rectangle to *aSize* and sends **textContainerChangedGeometry:** to the NSLayoutManager.

See also: – **setTextContainerInset:** (NSTextView), – **containerSize**

setHeightTracksTextView:

– (void)**setHeightTracksTextView:**(BOOL)*flag*

Controls whether the receiver adjusts the height of its bounding rectangle when its NSTextView is resized. If *flag* is YES, the receiver follows changes to the height of its text view; if *flag* is NO, it doesn't.

See the class description for more information on size tracking.

See also: – **setContainerSize:**, – **setWidthTracksTextView:**, – **heightTracksTextView**

setLayoutManager:

– (void)**setLayoutManager:**(NSLayoutManager *)*aLayoutManager*

Sets the receiver's NSLayoutManager to *aLayoutManager*. This method is invoked automatically when you add an NSTextContainer to an NSLayoutManager; you should never need to invoke it directly, but might want to override it. If you want to replace the NSLayoutManager for an established group of text-system objects, use **replaceLayoutManager:**.

See also: – **addTextContainer:** (NSLayoutManager), – **layoutManager**

setLineFragmentPadding:

– (void)**setLineFragmentPadding:**(float)*aFloat*

Sets the amount (in points) by which text is inset within line fragment rectangles to *aFloat*. Also sends **textContainerChangedGeometry:** to the receiver's NSLayoutManager to inform it of the change.

See also: – **lineFragmentRectForProposedRect:sweepDirection:movementDirection: remainingRect:**, – **lineFragmentPadding**

setTextView:

– (void)**setTextView:**(NSTextView *)*aTextView*

Sets the receiver's NSTextView to *aTextView* and sends **setTextContainer:** to *aTextView* to complete the association of the text container and text view. Since you usually specify an NSTextContainer when you create an NSTextView, you should rarely need to invoke this method. An NSTextContainer doesn't need an NSTextView to calculate line fragment rectangles, but must have one to display text.

You can use this method to disconnect an NSTextView from a group of text-system objects by sending this message to its text container and passing **nil** as *aTextView*.

See also: – **initWithFrame:textContainer:** (NSTextView), – **replaceTextContainer:** (NSTextView)

setWidthTracksTextView:

– (void)**setWidthTracksTextView:**(BOOL)*flag*

Controls whether the receiver adjusts the width of its bounding rectangle when its NSTextView is resized. If *flag* is YES, the receiver follows changes to the width of its text view; if *flag* is NO, it doesn't.

See the class description for more information on size tracking.

See also: – **setContainerSize:**, – **setHeightTracksTextView:**, – **widthTracksTextView**

textView

– (NSTextView *)**textView**

Returns the receiver's NSTextView, or **nil** if it has none.

See also: – **setTextView:**

widthTracksTextView

– (BOOL)**widthTracksTextView**

Returns YES if the receiver adjusts the width of its bounding rectangle when its NSTextView is resized, NO otherwise. The width is adjusted to the width of the NSTextView minus twice the inset width (as given by NSTextView's **textContainerInset** method).

See the class description for more information on size tracking.

See also: – **heightTracksTextView**, – **setWidthTracksTextView:**

NSTextField

Inherits From: NSControl : NSView : NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSTextField.h

Class Description

An NSTextField is a kind of NSControl that displays text that the user can select or edit, and which sends its action message to its target when the user presses the Return key while editing. Like other controls, it also performs validation on its value when edited; if the value isn't valid it sends a special error action message to its target. An NSTextField can be assigned a delegate, who is then informed of delegate messages sent by the window's field editor, such as **textShouldEndEditing:**. See the NSWindow and NSTextView class specifications for more information on a window's field editor.

Typical of several kinds of control, NSTextField allows you to set its text and background color, whether it draws the background, and whether it draws a bezel or border around its text. Note that the text and background colors of selected text are configurable. The selected text color overshadows any actual text color applied to the text while it's selected (this is generally the case with controls).

You can link text fields together in their window's key view loop, as described in the NSWindow class specification.

Method Types

Controlling editability and selectability

– setEditable:
– isEditable
– setSelectable:
– isSelectable

Setting the error action

– setErrorAction:
– errorAction

Controlling rich text behavior

– setAllowsEditingTextAttributes:
– allowsEditingTextAttributes
– setImportsGraphics:
– importsGraphics

Setting the text color

– setTextColor:
– textColor

Controlling the background

– setBackgroundColor:
– backgroundColor
– setDrawsBackground:
– drawsBackground

Setting a border

– setBezeled:
– isBezeled
– setBordered:
– isBordered

Linking text fields together

– setNextText:
– nextText
– setPreviousText:
– previousText

Selecting the text

– selectText:

Working with the responder chain

– acceptsFirstResponder

Using keyboard interface control

– setTitleWithMnemonic:

Setting the delegate

– setDelegate:
– delegate

Text delegate methods

– textShouldBeginEditing:
– textDidBeginEditing:
– textDidChange:
– textShouldEndEditing:
– textDidEndEditing:

Instance Methods

acceptsFirstResponder

– (BOOL)**acceptsFirstResponder**

Returns YES if the receiver is editable or selectable, NO otherwise.

allowsEditingTextAttributes

– (BOOL)**allowsEditingTextAttributes**

Returns YES if the receiver allows the user to change font attributes of the receiver's text, NO if the user isn't permitted to do so. You can change text attributes programmatically regardless of this setting.

See also: – **importsGraphics**, – **setAllowsEditingTextAttributes:**

backgroundColor

– (NSColor *)**backgroundColor**

Returns the color of the background that the receiver draws behind the text.

See also: – **drawsBackground**, – **setBackgroundColor:**

delegate

– (id)**delegate**

Returns the receiver's delegate.

See also: – **textShouldBeginEditing:**, – **textShouldEndEditing:**, – **textDidBeginEditing:**, – **textDidEndEditing:**, – **textDidChange:**, – **setDelegate:**

drawsBackground

– (BOOL)**drawsBackground**

Returns YES if the receiver's cell draws its background color behind its text, NO if it draws no background.

See also: – **backgroundColor**, – **drawsBackground** (NSTextFieldCell), – **setDrawsBackground:**

errorAction

– (SEL)**errorAction**

Returns the selector for the message sent to the receiver's target whenever a validation error occurs.

See also: – **textShouldEndEditing:**, – **setErrorAction:**

importsGraphics

– (BOOL)**importsGraphics**

Returns YES if the receiver allows the user to drag image files into it, NO if it doesn't accept dragged images. You can add images programmatically regardless of this setting.

See also: – **allowsEditingTextAttributes**, – **importsGraphics** (NSTextView), – **setImportsGraphics:**

isBezeled

– (BOOL)**isBezeled**

Returns YES if the receiver draws a bezeled frame around its contents, NO if it doesn't.

See also: – **isBordered**, – **setBezeled:**

isBordered

– (BOOL)**isBordered**

Returns YES if the receiver draws a solid black border around its contents, NO if it doesn't.

See also: – **isBezeled**, – **setBordered:**

isEditable

– (BOOL)**isEditable**

Returns YES if the user is allowed to select and edit the receiver's text, NO if the user isn't allowed to edit it (though the user may be able to select it).

See also: – **isSelectable**, – **setEditable:**

isSelectable

– (BOOL)**isSelectable**

Returns YES if the user is allowed to select the receiver's text, NO if the user isn't allowed to select it. Selectable text isn't necessarily editable; use **isEditable** to check for editability.

See also: – **setSelectable:**

nextText

– (id)**nextText**

Returns the receiver's next key view, the object that's made first responder when the user presses the Tab key while editing the receiver. See the description of the **nextKeyView** method in NSView for more information.

See also: – **previousText**, – **setNextText:**

previousText

– (id)**previousText**

Returns the receiver's previous key view, the object that's made first responder when the user presses Shift-Tab while editing the receiver. See the description of the **previousKeyView** method in NSView for more information.

See also: – **nextText**, – **setPreviousText:**

selectText:

– (void)**selectText:**(id)*sender*

Selects the entire contents of the receiver if it's selectable. However, if the receiver isn't in some window's view hierarchy, this method has no effect.

See also: – **isSelectable**

setAllowsEditingTextAttributes:

– (void)**setAllowsEditingTextAttributes:**(BOOL)*flag*

Controls whether the receiver allows the user to change font attributes of the receiver's text. If *flag* is YES, the user is permitted to make such changes; if *flag* is NO, the user isn't so permitted. You can change text attributes programmatically regardless of this setting.

See also: – **setImportsGraphics:**, – **allowsEditingTextAttributes**

setBackgroundColor:

– (void)**setBackgroundColor:**(NSColor *)*aColor*

Sets the color of the background that the receiver draws behind the text to *aColor*.

See also: – **setDrawsBackground:**, – **backgroundColor**

setBezeled:

– (void)**setBezeled:**(BOOL)*flag*

Controls whether the receiver draws a bezeled border around its contents. If *flag* is NO, it draws no border; if *flag* is YES, it draws a bezeled border and invokes **setDrawsBackground:** with an argument of NO.

See also: – **isBezeled**, – **setBordered:**

setBordered:

– (void)**setBordered:**(BOOL)*flag*

Controls whether the receiver draws a solid black border around its contents. If *flag* is YES, then it draws a border; if *flag* is NO, it draws no border.

See also: – **isBordered**, – **setBezeled:**

setDelegate:

– (void)**setDelegate:**(id)*anObject*

Sets the receiver's delegate to *anObject*.

See also: – **textShouldBeginEditing:**, – **textShouldEndEditing:**, – **textDidBeginEditing:**, – **textDidEndEditing:**, – **textDidChange:**, – **delegate**

setDrawsBackground:

– (void)**setDrawsBackground:**(BOOL)*flag*

Controls whether the receiver draws its background color behind its text. If *flag* is YES, then it does; if *flag* is NO, then it draws nothing behind its text.

See also: – **setBackgroundColor:**, – **setDrawsBackground:** (NSTextFieldCell), – **drawsBackground**

setEditable:

– (void)**setEditable:**(BOOL)*flag*

Controls whether the user can edits the receiver's text. If *flag* is YES, then the user is allowed to both select and edit text. If *flag* is NO, then the user isn't permitted to edit text, and the receiver's selectability is restored to its previous value. For example, if an NSTextField is selectable but not editable, then made editable for a time, then made not editable, it remains selectable. To guarantee that text is neither editable nor selectable, simply use **setSelectable:** to turn off selectability.

See also: – **isEditable**

setErrorAction:

– (void)**setErrorAction:**(SEL)*aSelector*

Sets the selector for the message sent to the receiver's target whenever a validation error occurs to *aSelector*.

See also: – **textShouldEndEditing:**, – **errorAction**

setImportsGraphics:

– (void)**setImportsGraphics:**(BOOL)*flag*

Controls whether the receiver allows the user to drag image files into it. If *flag* is YES, the receiver accepts dragged images; if *flag* is NO, it doesn't. You can add images programmatically regardless of this setting.

See also: – **setAllowsEditingTextAttributes:**, – **setImportsGraphics:** (NSTextView),
– **importsGraphics**

setNextText:

– (void)**setNextText:**(id)*anObject*

Sets the receiver's next key view to *anObject*, which should be a kind of NSView. See the description of the **setNextKeyView:** method in the NSView class specification for more information.

See also: – **setPreviousText:**, – **nextText**

setPreviousText:

– (void)**setPreviousText:**(id)*anObject*

Sets the receiver's previous key view to *anObject*, which should be a kind of NSView. See the description of the **setPreviousKeyView:** method in the NSView class specification for more information.

See also: – **setNextText:**, – **previousText**

setSelectable:

– (void)**setSelectable:**(BOOL)*flag*

If *flag* is YES, the receiver is made selectable but not editable (use **setEditable:** to make text both selectable and editable). If NO, then the text is made neither editable nor selectable.

See also: – **setEditable:**

setTextColor:

– (void)**setTextColor:**(NSColor *)*aColor*

Sets the color used to draw the receiver's text to *aColor*.

See also: – **setBackgroundColor:**, **– setTextColor:** (NSTextFieldCell), **– textColor**

setTitleWithMnemonic:

– (void)**setTitleWithMnemonic:**(NSString *)*aString*

Sets the receiver's string value to *aString*, using the first character preceded by an ampersand ('&') as the mnemonic and stripping out that first ampersand character. Use this method only with a non-editable text field being used as a label for another interface component, which you should establish using **setNextKeyView:**. When set up in this fashion, the text field's mnemonic serves to select the other interface component.

textColor

– (NSColor *)**textColor**

Returns the color used to draw the receiver's text.

See also: – **backgroundColor**, – **textColor** (NSTextFieldCell), – **setTextColor:**

textDidBeginEditing:

– (void)**textDidBeginEditing:**(NSNotification *)*aNotification*

Posts an NSControlTextDidBeginEditingNotification to the default notification center. This causes the receiver's delegate to receive a **controlTextDidBeginEditing:** message. See the NSControl class specification for more information on the text delegate method.

See also: – **textDidBeginEditing:**, – **textDidChange:**, – **textShouldEndEditing:**, – **textDidEndEditing:**

textDidChange:

– (void)**textDidChange:**(NSNotification *)*aNotification*

Forwards this message to the receiver's cell if it responds, and posts an NSControlTextDidChangeNotification to the default notification center. This causes the receiver's delegate to receive a **controlTextDidChange:** message. See the NSControl class specification for more information on the text delegate method.

See also: – **textShouldBeginEditing:**, – **textDidBeginEditing:**, – **textShouldEndEditing:**, – **textDidEndEditing:**

textDidEndEditing:

– (void)**textDidEndEditing:**(NSNotification *)*aNotification*

Handles an end to editing. After validating the new value, posts an NSControlTextDidEndEditingNotification to the default notification center. This causes the receiver's delegate to receive a **controlTextDidEndEditing:** message. After this, sends **endEditing:** to the receiver' cell, and handles the key that caused editing to end:

- If the user ended editing by pressing Return, this method tries to send the receiver's action to its target; if unsuccessful, it sends **performKeyEquivalent:** to its NSWindow (for example, to handle the default button on a panel); if that also fails, then the receiver simply selects its text.
- If the user ended editing by pressing Tab or Shift-Tab, the receiver tries to have its NSWindow select its next or previous key view, using the NSWindow method **selectKeyViewFollowingView:** or **selectKeyViewPreceedingView:**. If unsuccessful in doing this, the receiver simply selects its text.

See the NSControl class specification for more information on the text delegate method.

See also: – **textShouldBeginEditing:**, – **textDidBeginEditing:**, – **textDidChange:**, – **textShouldEndEditing:**

textShouldBeginEditing:

– (BOOL)**textShouldBeginEditing:**(NSText *)*textObject*

If the receiver isn't editable, returns NO immediately. If it is editable and its delegate responds to **control: textShouldBeginEditing:**, invokes that method and returns the result. Otherwise simply returns YES to allow editing to occur. See the NSControl class specification for more information on the text delegate method.

See also: – **textDidBeginEditing:**, – **textDidChange:**, – **textShouldEndEditing:**, – **textDidEndEditing:**

textShouldEndEditing:

– (BOOL)**textShouldEndEditing:**(NSText *)*textObject*

Performs validation on the receiver's new value using NSCell's **isEntryAcceptable:**, sending the receiver's error action to its target if validation fails. If the new value is valid and the delegate responds to **control: textShouldEndEditing:**, invokes that method and returns the result, in addition beeping if the delegate returns NO. See the NSControl class specification for more information on the text delegate method.

See also: – **textShouldBeginEditing:**, – **textDidBeginEditing:**, – **textDidChange:**, – **textDidEndEditing:**, – **errorAction**

NSTextFieldCell

Inherits From: NSActionCell : NSCell : NSObject

Conforms To: NSCoding (NSCell)
NSCopying (NSCell)
NSObject (NSObject)

Declared In: AppKit/NSTextFieldCell.h

Class Description

NSTextFieldCell adds to NSCell's text-display capabilities by allowing you to set the color of both the text and its background. You can also specify whether the cell draws its background at all. All of the methods declared by this class are also declared by NSTextField, which uses NSTextFieldCells to draw and edit text.

Method Types

Setting the text color
– setTextColor:
– textColor

Controlling the background
– setBackgroundColor:
– backgroundColor
– setDrawsBackground:
– drawsBackground

Changing the field editor
– setUpFieldEditorAttributes:

Instance Methods

backgroundColor

– (NSColor *)**backgroundColor**

Returns the color of the background that the receiver draws behind the text.

See also: – **drawsBackground**, – **backgroundColor** (NSTextField), – **setBackgroundColor:**

drawsBackground

– (BOOL)**drawsBackground**

Returns YES if the receiver's cell draws its background color behind its text, NO if it draws no background.

See also: – **backgroundColor**, – **drawsBackground** (NSTextField), – **setDrawsBackground:**

setBackgroundColor:

– (void)**setBackgroundColor:**(NSColor *)*color*

Sets the color of the background that the receiver draws behind the text to *aColor*.

See also: – **setDrawsBackground:**, – **setBackgroundColor:** (NSTextField) – **backgroundColor**

setDrawsBackground:

– (void)**setDrawsBackground:**(BOOL)*flag*

Controls whether the receiver draws its background color behind its text. If *flag* is YES, then it does; if *flag* is NO, then it draws nothing behind its text.

See also: – **setBackgroundColor:**, – **setDrawsBackground:** (NSTextField), – **drawsBackground**

setTextColor:

– (void)**setTextColor:**(NSColor *)*color*

Sets the color used to draw the receiver's text to *aColor*.

See also: – **setBackgroundColor:**, **– setTextColor:** (NSTextField), **– textColor**

setUpFieldEditorAttributes:

– (NSText *)**setUpFieldEditorAttributes:**(NSText *)*textObj*

You never invoke this method directly; by overriding it, however, you can customize or replace the field editor. When you override this method, you should generally invoke super, and return the *textObj* argument. For information on field editors, see the "Field Editors" section of the NSWindow class description.

textColor

– (NSColor *)**textColor**

Returns the color used to draw the receiver's text.

See also: – **backgroundColor**, – **textColor** (NSTextField), – **setTextColor:**

NSTextStorage

Inherits From: NSMutableAttributedString : NSAttributedString : NSObject

Conforms To: NSCopying
NSMutableCopying
NSObject (NSObject)

Declared In: AppKit/NSTextStorage.h

Class Description

NSTextStorage is a semi-concrete subclass of NSMutableAttributedString that manages a set of client NSLayoutManagers, notifying them of any changes to its characters or attributes so that they can re-lay and redisplay the text as needed. NSTextStorage defines the fundamental storage mechanism of NeXT's extended text-handling system.

Like an abstract class of a class cluster, allocating and initializing an NSTextStorage actually produces an instance of a private subclass. You can use any of NSAttributedString and NSMutableAttributedString's initialization methods to create an NSTextStorage object. Following this, you add NSLayoutManagers to it using **addLayoutManager:**.

The behavior of an NSTextStorage object is best illustrated by following the methods it invokes while being changed. There are three stages to editing a text storage object programmatically. The first stage is to send it a **beginEditing** message to announce a group of changes. In the second stage, you send it some editing messages, such as **deleteCharactersInRange:** and **addAttributes:range:**, to effect the changes in characters or attributes. Each time you send such a method, the text storage object invokes **edited:range: changeInLength:** to record the range of its characters affected since it received the **beginEditing** message. For the third stage, when you're done changing the text storage object, you send it an **endEditing** message. This causes it to invoke its own **processEditing** method, fixing attributes within the recorded range of changed characters. After fixing its attributes, the text storage object sends a message to each NSLayoutManager indicating the range in the text storage object that has changed, along with the nature of those changes. The NSLayoutManagers in turn use this information to re-lay their glyphs and redisplay if necessary. NSTextStorage also keeps a delegate and sends it messages before and after processing edits.

Creating a Subclass of NSTextStorage

As indicated above, NSTextStorage isn't a fully concrete class. It defines the storage for its NSLayoutManagers and implements all of the methods described in this specification, but doesn't provide the primitive attributed string methods to subclasses. A subclass must define the storage for its attributed string, typically as an instance variable of type NSMutableAttributedString, override **init** and define its own

initialization methods, and implement the primitive methods of both NSAttributedString and NSMutableAttributedString. For the record, these methods are:

– string
– attributesAtIndex:effectiveRange:
– replaceCharactersInRange:withString:
– setAttributes:range:

Beyond these requirements, if a subclass overrides or adds any methods that change its characters or attributes directly (not using the primitive methods or making extra changes after invoking the primitives), those methods must invoke **edited:range:changeInLength:** after performing the change in order to keep the change-tracking information up to date. See the description of this method for more information.

Method Types

Managing NSLayoutManagers
– addLayoutManager:
– removeLayoutManager:
– layoutManagers

Handling text edited messages
– edited:range:changeInLength:
– endEditing
– processEditing

Determining the nature of changes
– editedMask

Determining the extent of changes
– editedRange
– changeInLength

Setting the delegate
– setDelegate:
– delegate

Instance Methods

addLayoutManager:

– (void)**addLayoutManager:**(NSLayoutManager *)*aLayoutManager*

Adds *aLayoutManager* to the receiver's set of NSLayoutManagers.

See also: – **removeLayoutManager:**, – **layoutManagers**

changeInLength

– (int)**changeInLength**

Returns the difference between the current length of the edited range and its length before editing began (that is, before the receiver was sent the first **beginEditing** message or a single **edited:range: changeInLength:** message). This difference is accumulated with each invocation of **edited:range: changeInLength:**, until a final **endEditing** message processes the changes.

The receiver's delegate and layout managers can use this information to determine the nature of edits in their respective notification methods.

See also: – **editedRange**, – **editedMask**

delegate

– (id)**delegate**

Returns the receiver's delegate.

See also: – **setDelegate:**

edited:range:changeInLength:

– (void)**edited:**(unsigned)*mask*
range:(NSRange)*oldRange*
changeInLength:(int)*lengthChange*

Tracks changes made to the receiver, allowing the NSTextStorage to record the full extent of changes made between a pair of **beginEditing** and **endEditing** messages. If invoked outside of such a pair, this method immediately invokes **processEditing**. NSTextStorage invokes this method automatically each time it makes a change to its attributed string. Subclasses that override or add methods that alter their attributed strings directly should invoke this method after making those changes. The information accumulated with this method is then used in an invocation of **processEditing** to report the affected portion of the receiver.

mask specifies the nature of the changes. Its value is made by combining these options with the C bitwise OR operator:

Option	Meaning
NSTextStorageEditedAttributes	Attributes were added, removed, or changed.
NSTextStorageEditedCharacters	Characters were added, removed, or replaced.

oldRange indicates the extent of characters affected before the change took place. If the NSTextStorageEditedCharacters bit of *mask* is set, *lengthChange* gives the number of characters added to or removed from *oldRange* (otherwise its value is irrelevant). For example, when replacing “The” with “Several” in the string “The files couldn’t be saved”, *oldRange* is {0, 3} and *lengthChange* is 4.

Note: The methods for querying changes, **editedRange** and **changeInLength**, indicate the extent of characters affected *after* the change. This method expects the characters before the change because that information is readily available as the argument to whatever method performs the change (such as **replaceCharactersInRange:withString:**).

editedMask

– (unsigned int)**editedMask**

Returns the kinds of edits pending for the receiver, as a mask containing either or both of NSTextStorageEditedAttributes and NSTextStorageEditedCharacters. Use the C bitwise AND operator to test the mask; testing for equality will fail if additional mask flags are added later. The receiver’s delegate and layout managers can use this information to determine the nature of edits in their respective notification methods.

See also: – **editedRange**, – **changeInLength**

editedRange

– (NSRange)**editedRange**

Returns the range of the receiver to which pending changes have been made, whether of characters or of attributes. The receiver’s delegate and layout managers can use this information to determine the nature of edits in their respective notification methods.

See also: – **changeInLength**, – **editedMask**

endEditing

– (void)**endEditing**

Clears the last recorded invocation of **beginEditing**, and if there are no more, invokes **processEditing** to clean up after changes and notify the delegate and layout managers of the edits.

layoutManagers

– (NSArray *)**layoutManagers**

Returns the receiver's NSLayoutManagers.

See also: – **addLayoutManager:**, – **removeLayoutManager:**

processEditing

– (void)**processEditing**

Cleans up changes made to the receiver and notifies its delegate and layout managers of changes. This method is automatically invoked in response to an **endEditing** or **edited:range:changeInLength:** message. You should never need to invoke it directly.

This method begins by posting an NSTextStorageWillProcessEditingNotification to the default notification center (which results in the delegate receiving a **textStorageWillProcessEditing:** message). It then invokes the inherited **fixAttributesAfterEditingRange:** method to fix up attributes after a batch of editing changes. After this, it posts an NSTextStorageDidProcessEditingNotification to the default notification center (which results in the delegate receiving a **textStorageDidProcessEditing:** message). Finally, it sends a **textStorage:edited:range:changeInLength:invalidatedRange:** message to each of the receiver's NSLayoutManagers using the argument values provided.

removeLayoutManager:

– (void)**removeLayoutManager:**(NSLayoutManager *)*aLayoutManager*

Removes *aLayoutManager* from the receiver's set of NSLayoutManagers.

See also: – **addLayoutManager:**, – **layoutManagers**

setDelegate:

– (void)**setDelegate:**(id)*anObject*

Sets the receiver's delegate to *anObject*.

See also: – **delegate**

Methods Implemented By the Delegate

textStorageDidProcessEditing:

– (void)**textStorageDidProcessEditing:**(NSNotification *)*aNotification*

Informs the delegate that an NSTextStorage object has finished processing edits. The text storage object is available by sending **object** to *aNotification*, which is always an NSTextStorageDidProcessEditingNotification. The delegate can use this notification to verify the final state of the text storage object; it can't change the text storage object's characters without leaving it in an inconsistent state, but if necessary it can change attributes. Note that even in this case it's possible to put a text storage object into an inconsistent state—for example by changing the font of a range to one that doesn't support the characters in that range (such as using a Latin font for Kanji text).

textStorageWillProcessEditing:

– (void)**textStorageWillProcessEditing:**(NSNotification *)*aNotification*

Informs the delegate that an NSTextStorage object is about to process edits. The text storage object is available by sending **object** to *aNotification*, which is always an NSTextStorageWillProcessEditingNotification. The delegate can use this notification to verify the changed state of the text storage object, and to make changes to the text storage object's characters or attributes to enforce whatever constraints it establishes (which doesn't result in this message being sent again, however). For example, a code editor application might add a delegate that checks after edits to make sure that all programming language keywords are set in boldface.

Notifications

NSTextStorageDidProcessEditingNotification

Posted after the NSTextStorage finishes processing edits in **processEditing**. Observers other than the delegate shouldn't make further changes to the NSTextStorage. This notification contains a notification object but no userInfo dictionary. The notification object is the NSTextStorage object that processed the edits.

NSTextStorageWillProcessEditingNotification

Posted before the NSTextStorage finishes processing edits in **processEditing**. Observers other than the delegate shouldn't make further changes to the NSTextStorage. This notification contains a notification object but no userInfo dictionary. The notification object is the NSTextStorage object that is a bout to process the edits.

NSTextTab

Inherits From:	NSObject
Conforms To:	NSCopying NSObject (NSObject)
Declared In:	AppKit/NSParagraphStyle.h

Class Description

An NSTextTab represents a tab in an NSParagraphStyle object, storing an alignment type and location. NSTextTabs are most frequently used with the Application Kit's text system and with NSRulerView and NSRulerMarker objects. See the appropriate class specifications for more information on these uses.

The text system supports four alignment types: left, center, right, and decimal (based on the decimal separator character of the locale in effect). These alignment types are absolute, not based on the line sweep direction of text. For example, tabbed text is always positioned to the left of a right-aligned tab, whether the line sweep direction is left-to-right or right-to-left. A tab's location, on the other hand, is relative to the back margin. A tab set at 1.5", for example, is at 1.5" from the right in right-to-left text.

Adopted Protocols

NSCopying

– copyWithZone:

Method Types

Creating an NSTextTab

– initWithType:location:

Getting tab stop information

– location
– tabStopType

Instance Methods

initWithType:location:

– (id)**initWithType:**(NSTextTabType)*type* **location:**(float)*location*

Initializes a newly allocated NSTextTab with an alignment of *type* at *location* on the paragraph. The location is relative to the back margin, based on the line sweep direction of the paragraph. *type* can be any one of:

NSLeftTabStopType	NSRightTabStopType
NSCenterTabStopType	NSDecimalTabStopType

location

– (float)**location**

Returns the receiver's ruler location relative to the back margin.

tabStopType

– (NSTextTabType)**tabStopType**

Returns the receiver's tab stop type. The possible values are listed in the **initWithType:location:** method description.

NSTextView

Inherits From: NSText : NSView : NSResponder : NSObject

Conforms To: NSTextInput
NSChangeSpelling (NSText)
NSIgnoreMisspelledWords (NSText)
NSCoding (NSResponder) — *Note: NSTextView doesn't implement this protocol*
NSObject (NSObject)

Declared In: AppKit/NSTextView.h

Class at a Glance

Purpose

NSTextView is the front-end component of the Application Kit's text system. It displays and manipulates text laid out in an area defined by an NSTextContainer, and adds many features to those defined by its superclass,

NSText. Many of the methods that you'll use most frequently are declared by the superclass; see the NSText class specification for details.

Principal Attributes

- Supports rich text and graphics
- Supports input management and key bindings
- Works with the Font Panel and menu
- Works with rulers
- Provides delegation and notification
- Works with the Services facility
- Works with the pasteboard
- Works with spell-checking services

Creation

Instances of this class can be created using Interface Builder or using one of the following methods:

– initWithFrame:	Creates an NSTextView along with all its supporting objects.
– initWithFrame:textContainer:	Designated initializer.

Commonly Used Methods

The methods most commonly used with NSTextView objects are declared in NSText, the superclass. These methods provide access to the other major components of the text system:

– textStorage	Returns the associated NSTextStorage object.
– textContainer	Returns the associated NSTextContainer object.
– layoutManager	Returns the associated NSLayoutManager object.

Class Description

NSTextView is the front-end class to the Application Kit's extended text-handling system. It draws the text managed by the back-end components and handles user events to select and modify its text. NSTextView is the principal means to obtain a text object that caters to almost all needs for displaying and managing text at the user interface level. While NSTextView is a subclass of NSText—which declares the most general OpenStep interface to the text system—NSTextView adds several major features over and above the capabilities of NSText.

One of the design goals of NSTextView is to provide a comprehensive set of text-handling features so that you should rarely need to create a subclass. In its standard incarnation, NSTextView creates the requisite group of objects that support the text handling system—NSTextContainer, NSLayoutManager, and NSTextStorage objects. Refer to "The OPENSTEP Text System" for a comprehensive overview of the components of the text system. Here are the major features that NSTextView adds to those of NSText:

Rulers. NSTextView works with the NSRulerView class to let users control paragraph formatting, in addition to using commands in the Format Text menu provided by Interface Builder.

Input management and key binding. Certain key combinations are bound to specific NSTextView methods so that the user can move the insertion point, for example, without using the mouse.

Marked text attributes. NSTextView defines a set of text attributes that support special display characteristics during input management. Marked text attributes only affect visual aspects of text—color, underline, and so on—they don't include any attributes that would change the layout of text.

File and graphic attachments. The extended text system provides programmatic access to text attachments as instances of NSTextAttachment, through the NSTextView and NSTextStorage classes.

Delegate messages and notifications. NSTextView adds several delegate messages and notifications to those used by NSText. The delegate and observers of an NSTextView can receive any of the messages or notifications declared by both classes.

Creating NSTextView Objects

The easiest way to add an NSTextView to your application is through Interface Builder. Interface Builder's Data Views palette supplies a specially configured NSScrollView object that contains an NSTextView object as its document view. This NSTextView is configured to work with the NSScrollView and other user-interface controls such as a ruler, the Font menu, the Edit menu, and so on.

Interface Builder also offers other objects—of the NSTextField and NSForm classes—that make use of NSTextView objects for their text-editing facilities. In fact, all NSTextFields and NSForms within the same window share the same NSTextView object (known as the *field editor*), thus reducing the memory demands of an application. If your application requires stand-alone or grouped text fields that support editing (and all the other facilities provided by the NSTextView class), these are the classes to use.

You can also create NSTextView objects programmatically, using either of the methods **initWithFrame: textContainer:** (the designated initializer), or **initWithFrame:**. The **initWithFrame:** method is the simplest way to obtain an NSTextView object—it creates all the other components of the text-handling system for you and releases them when you're done. If you use **initWithFrame:textContainer:**, you must construct (and release) the other components yourself. See the "The OPENSTEP Text System" for more information.

Configuring Editing Behavior

Like NSText, NSTextView allows you to grant or deny the user the ability to select or edit its text, using the **setSelectable:** and **setEditable:** methods. These methods only affect what the user can do; you can still make changes to the NSTextView programmatically. An editable text view can behave as a normal text editor, accepting Tab and Return characters, or as a field editor, interpreting tabs and returns as cues to end editing. The **setFieldEditor:** method controls this behavior. NSTextView also implements the distinction between plain and rich text defined by NSText with its **setRichText:** and **setImportsGraphics:** methods. See the NSText class specification for more information on these various distinctions.

Attachments

While NSText leaves open the nature of imported graphics and other attachments, NSTextView explicitly uses NSTextAttachment objects, which contain NSFileWrappers to represent the attached files. NSTextView declares several delegate methods that let you handle user actions on an attachment's image or icon. **textView:clickedOnCell:inRect:atIndex:** and **textView:doubleClickedOnCell:inRect:atIndex:** let the delegate take action on mouse clicks, **textView:draggedCell:inRect:event:atIndex:** lets the delegate initiate a dragging session for the attachment, and **textView:clickedOnLink:atIndex:** lets the delegate act on mouse clicks in a link. See the NSTextAttachment, NSTextAttachmentCell, and NSFileWrapper class and protocol specifications for more information on working with attachments.

Input Management

NSTextView uses an input manager to turn basic character information into text and commands. It passes uninterpreted keyboard input to the input manager, which examines the characters generated and sends messages to the NSTextView based on those characters. If the typed characters are interpreted as text to input, the input manager sends the text view an **insertText:** message. If they're interpreted as commands to perform, such as moving the insertion point or deleting text, the input manager sends the text view a **doCommandBySelector:** message. Many of the standard commands are described in the NSResponder class specification. NSTextView also gives its delegate a chance to handle a command by sending it a **textView:doCommandBySelector:** message. If the delegate implements this method and returns YES, the text view does nothing further; otherwise it tries to perform the command itself.

See the NSInputManager class and NSTextInput protocol specifications for more information.

Using the Font Panel and Ruler

NSTextView is designed to work with the Application Kit's font conversion system, defined by the NSFontPanel and NSFontManager classes. By default, an NSTextView keeps the Font Panel updated with the first font in its selection, or of its typing attributes (defined below). It also changes the font in response to messages from the Font Panel and Font menu. Such changes apply to the selected text or typing attributes for a rich text view, or to all the text in a plain text view. You can turn this behavior off using the **setUsesFontPanel:** method. Doing so is recommended for a text view that serves as a field editor, for example. Making an NSTextView not use the font conversion system renders some of its other methods

unusable, as these methods require access to font information to work. See the description of **setUsesFontPanel:** for these side effects.

NSTextView also defines a comprehensive interface for manipulating paragraph attributes, using the NSRulerView class. If an NSTextView is enclosed in an NSScrollView, it can display a ruler view, which displays margin and tab markers that the user can manipulate to adjust their settings, as well as other controls for setting alignment, paragraph spacing, and so on. **setRulerVisible:** and the inherited **toggleRuler:** control whether the ruler view is displayed. The NSTextView serves as the ruler view's client, as described in the NSRulerView class specification. Similar to the Font Panel, NSTextView can be set not to use a ruler with the **setUsesRuler:** method. This has side effects similar to those of **setUsesFontPanel:**.

Examining and Setting the Selection

Most of the time the selection is determined by the user through mouse or keyboard operations. You can get the range of characters currently selected using the **selectedRange** method. This is the single most commonly used method for examining the selection. You can also set the selection programmatically using **setSelectedRange:**. NSTextView indicates its selection by applying a special set of attributes to it. **selectedTextAttributes** returns these attributes, and **setSelectedTextAttributes:** sets them.

While changing the selection in response to user input, an NSTextView invokes its **setSelectedRange: affinity:stillSelecting:** method. The first argument is of course the range to select. The second, called the selection affinity, determines which glyph the insertion point displays near when the two glyphs aren't adjacent. It's typically used where lines wrap to place the insertion point at the end of one line or the beginning of the following line. You can get the selection affinity in effect using the **selectionAffinity** method. The last argument indicates whether the selection is still in the process of changing; the delegate and any observers aren't notified of the change in the selection until the method is invoked with NO for this argument. An additional factor affecting selection behavior is the selection granularity: whether characters, words, or whole paragraphs are being selected. This is usually determined by number of initial clicks; for example, a double-click initiates word-level selection. NSTextView decides how much to change the selection during input tracking using its **selectionRangeForProposedRange:granularity:** method, as described under "Subclass Responsibilities" below.

An additional aspect of selection, actually related to input management, is the range of marked text. As the input manager interprets keyboard input, it can mark incomplete input in a special way. **markedRange** returns the range of any marked text, and **markedTextAttributes** returns the attributes used to highlight the marked text. You can change these attributes using **setMarkedTextAttributes:**

Setting Text Attributes

NSTextView allows you to change the attributes of its text programmatically through various methods, most inherited from the superclass, NSText. NSTextView adds its own methods for setting the attributes of text that the user types, for setting the baseline offset of text as an absolute value, and for adjusting kerning and use of ligatures. Most of the methods for changing attributes are defined as action methods, and apply to the selected text or typing attributes for a rich text view, or to all of the text in a plain text view.

An NSTextView maintains a set of *typing attributes* (font, size, color, and so on) that it applies to newly entered text, whether typed by the user or pasted as plain text. It automatically sets the typing attributes to the attributes of the first character immediately preceding the insertion point, of the first character of a paragraph if the insertion point is at the beginning of a paragraph, or of the first character of a selection. The user can change the typing attributes by choosing menu commands and using utilities such as the Font Panel. You can also set the typing attributes programmatically using **setTypingAttributes:**, though you should rarely find need to do so unless creating a subclass.

NSText defines the action methods **superscript:**, **subscript:**, and **unscript:**, which raise and lower the baseline of text by predefined increments. NSTextView gives you much finer control over the baseline offset of text by defining the **raiseBaseline:** and **lowerBaseline:** action methods, which raise or lower text by one point each time they're invoked.

Kerning

NSTextView provides convenient action methods for adjusting the spacing between characters. By default, an NSTextView object uses standard kerning (as provided by the data in a font's AFM file). A **turnOffKerning:** message causes this kerning information to be ignored and the selected text to be displayed using nominal widths. The **loosenKerning:** and **tightenKerning:** methods adjust kerning values over the selected text and **useStandardKerning:** reestablishes the default kerning values.

Kerning information is a character attribute that's stored in the text view's NSTextStorage object. If your application needs finer control over kerning than the methods of this class provide, you should operate on the NSTextStorage object directly through methods defined by its superclass, NSMutableAttributedString. See the NSAttributedString Class Cluster Additions specification for information on setting attributes.

Ligatures

NSTextView's support for ligatures provides the minimum required ligatures for a given font and script. The required ligatures for a specific font and script are determined by the mechanisms that generate glyphs for a specific language. Some scripts may well have no ligatures at all—English text, as an example, doesn't require ligatures, although certain ligatures such as "fi" and "fl" are desirable and are used if they're available. Other scripts, such as Arabic, demand that certain ligatures must be available even if a **turnOffLigatures:** message is sent to the NSTextView. Other scripts and fonts have standard ligatures that are used if they're available. The **useAllLigatures:** method extends ligature support to include all possible ligatures available in each font for a given script.

Ligature information is a character attribute that's stored in the text view's NSTextStorage object. If your application needs finer control over ligature use than the methods of this class provide, you should operate on the NSTextStorage object directly through methods defined by its superclass, NSMutableAttributedString. See the NSAttributedString Class Cluster Additions specification for information on setting attributes.

Using Multiple NSTextViews

A single NSLayoutManager can be assigned any number of NSTextContainers, in whose NSTextViews it lays out text sequentially. In such a configuration, many of the attributes accessed through the NSTextView interface are actually shared by all of these text views. Among these attributes are:

- The selection
- The delegate (see “Other Delegate Messages and Notifications” below for details)
- Selectability
- Editability
- Whether they act as a field editor
- Whether they display plain or rich text
- Whether they import graphics
- Whether the ruler is visible
- Whether they use the Font Panel
- Whether they use the ruler

Setting any of these attributes causes all associated NSTextView’s to share the new value.

With multiple NSTextViews, only one is the first responder at any time. NSLayoutManager defines these methods for determining and appropriately setting the first responder:

– layoutManagerOwnsFirstResponderInWindow:
– firstTextView
– textViewForBeginningOfSelection

See their descriptions in the NSLayoutManager class specification for more information.

Other Delegate Messages and Notifications

An NSTextView object can have a delegate that it informs of certain actions or pending changes to the state of the text. Several of the delegate methods have already been mentioned; here are all of the messages that a delegate can receive:

textView:willChangeSelectionFromCharacterRange:toCharacterRange:
textViewDidChangeSelection:

textShouldBeginEditing:
textDidBeginEditing:
textView:shouldChangeTextInRange:replacementString:
textDidChange:
textShouldEndEditing:
textDidEndEditing:

textView:doCommandBySelector:

textView:clickedCell:inRect:
textView:doubleClickedCell:inRect:
textView:draggedCell:inRect:event:

Those whose names begin with “text” rather than “textView” are declared by NSText and described in the NSText class specification. See “Methods Implemented By the Delegate” at the end of this class description for more details. The delegate can be any object you choose, and one delegate can control multiple NSTextView objects (or multiple series of connected NSTextView objects).

All NSTextView objects attached to the same NSLayoutManager share the same delegate: Setting the delegate of one such NSTextView sets the delegate for all the others. Delegate messages pass the **id** of the sender as an argument. For multiple NSTextViews attached to the same NSLayoutManager, the **id** is that of the *notifying text view*, the first NSTextView for the shared NSLayoutManager. As the name implies, this NSTextView is also responsible for posting notifications at the appropriate times.

The notifications posted by NSTextView are:

NSTextViewDidChangeSelectionNotification

NSTextDidBeginEditingNotification
NSTextDidEndEditingNotification
NSTextDidChangeNotification

NSTextViewWillChangeNotifyingTextViewNotification

Of these, the last is crucially import for observers to register for. If a new NSTextView is added at the beginning of a series of connected NSTextViews, it becomes the new notifying text view. It doesn’t have access to which objects are observing its group of text objects, so it posts an NSTextViewWillChangeNotifyingTextViewNotification, which allows all those observers to unregister themselves from the old notifying text view and reregister themselves with the new one. See the description for this notification at the end of this specification for more information.

Subclass Responsibilities

NSTextView expects subclasses to abide by certain rules of behavior, and provides many methods to help subclasses do so. Some of these methods are meant to be overridden to add information and behavior into the basic infrastructure. Some are meant to be invoked as part of that infrastructure when the subclass defines its own behavior. The following sections describe the major areas where a subclass has obligations or where it can expect help in implementing its new features.

Updating State

NSTextView automatically updates the Font Panel and ruler as its selection changes. If you add any new font or paragraph attributes to your subclass of NSTextView, you’ll need to override the methods that perform this updating to account for the added information. **updateFontPanel** makes the Font Panel display the font of the first character in the selection; you might override it to update the display of an

accessory view in the Font Panel. Similarly, **updateRuler** causes the ruler to display the paragraph attributes for the first paragraph in the selection. You can also override this to customize display of items in the ruler. Be sure to invoke **super**'s implementation to have the basic updating performed as well.

Custom Import Types

NSTextView supports pasteboard operations and the dragging of files and colors into its text. If you customize the ability of your subclass to handle pasteboard operations for new data types, you should override the **readablePasteboardTypes** and **writablePasteboardTypes** methods to reflect those types. Similarly, for to support new types of data for dragging operations, you should override the **acceptableDragTypes** method. Your implementation of these methods should invoke the superclass' implementation, add the new data types to the array returned from super, and then return the modified array.

For dragging operations, if your subclass's ability to accept your custom dragging types varies over time, you can override **updateDragTypeRegistration** to register or unregister the custom types according to the text view's current status. By default this method enables dragging of all acceptable types if the receiver is editable and a rich text view.

To read and write custom pasteboard types, you must override the **readSelectionFromPasteboard:type:** and **writeSelectionToPasteboard:type:** methods. In your implementation of these methods, you should read the new data types your subclass supports and let the superclass handle any other types.

Altering Selection Behavior

Your subclass of NSTextView can customize the way selections are made for the various granularities described in "Examining and Setting the Selection". While tracking user changes to the selection, whether by the mouse or keyboard, an NSTextView repeatedly invokes **selectionRangeForProposedRange: granularity:** to determine what range to actually select. When finished tracking changes, it sends the delegate a **textView:willChangeSelectionFromCharacterRange:toCharacterRange:**. message By overriding the NSTextView method or implementing the delegate method, you can alter the way the selection is extended or reduced. For example, in a code editor you can provide a delegate that extends a double click on a brace or parenthesis character to its matching delimiter.

Note: These mechanisms aren't meant for changing language word definitions (such as what's selected on a double click). This detail of selection is handled at a lower (and currently private) level of the text system.

Preparing to Change Text

If you create a subclass of NSTextView to add new capabilities that will modify the text in response to user actions, you may need to modify the range selected by the user before actually applying the change. For example, if the user is making a change to the ruler, the change must apply to whole paragraphs, so the selection may have to be extended to paragraph boundaries. Three methods calculate the range to which certain kinds of change should apply. **rangeForUserTextChange** returns the range to which any change to characters themselves—insertions and deletions—should apply.

rangeForUserCharacterAttributeChange returns the range to which a character attribute change, such as a new font or color, should apply. Finally, **rangeForUserParagraphAttributeChange** returns the range for a paragraph-level change, such as a new or moved tab stop, or indent. These methods all return a range whose location is NSNotFound if a change isn't possible; you should check the returned range and abandon the change in this case.

Notifying About Changes to the Text

In actually making changes to the text, you must ensure that the changes are properly performed and recorded by different parts of the text system. You do this by bracketing each batch of potential changes with **shouldChangeTextInRange:replacementString:** and **didChangeText** messages. These methods ensure that the appropriate delegate messages are sent and notifications posted. The first method asks the delegate for permission to begin editing with a **textShouldBeginEditing:** message. If the delegate returns NO, **shouldChangeTextInRange:replacementString:** in turn returns NO, in which case your subclass should disallow the change. If the delegate returns YES, the text view posts an NSTextDidBeginEditingNotification, and **shouldChangeTextInRange:replacementString:** in turn returns YES. In this case you can make your changes to the text, and follow up by invoking **didChangeText**. This method concludes the changes by posting an NSTextDidChangeNotification, which results in the delegate receiving a **textDidChange:** message.

The **textShouldBeginEditing:** and **textDidBeginEditing:** messages are sent only once during an editing session. More precisely, they're sent upon the first user input since the NSTextView became the first responder. Thereafter, these messages—and the NSTextDidBeginEditingNotification—are skipped in the sequence. **textView:shouldChangeTextInRange:replacementString:**, however, must be invoked for each individual change.

Smart Insert and Delete

NSTextView defines several methods to aid in "smart" insertion and deletion of text, so that spacing and punctuation is preserved after a change. Smart insertion and deletion typically applies when the user has selected whole words or other significant units of text. A smart deletion of a word before a comma, for example, also deletes the space that would otherwise be left before the comma (though not placing it on the pasteboard in a Cut operation). A smart insertion of a word between another word and a comma adds a space between the two words to protect that boundary. NSTextView automatically uses smart insertion and deletion by default; you can turn this behavior off using **setSmartInsertDeleteEnabled:**. Doing so causes only the selected text to be deleted, and inserted text to be added with no addition of white space.

If your subclass of NSTextView defines any methods that insert or delete text, you can make them smart by taking advantage of two NSTextView methods. **smartDeleteRangeForProposedRange:** expands a proposed deletion range to include any whitespace that should also be deleted. If you need to save the text deleted, though, it's typically best to save only the text from the original range. For smart insertion, **smartInsertForString:replacingRange:beforeString:afterString:** returns by reference two strings that you can insert before and after a given string to preserve spacing and punctuation. See the method descriptions for more information.

Adopted Protocols

NSTextInput

– conversationIdentifier
– doCommandBySelector:
– getMarkedText:selectedRange:
– hasMarkedText
– insertText:
– setMarkedText:selectedRange:
– unmarkText

Method Types

Creating an instance

– initWithFrame:textContainer:
– initWithFrame:

Registering Services information

+ registerForServices

Accessing related text-system objects

– setTextContainer:
– replaceTextContainer:
– textContainer
– setTextContainerInset:
– textContainerInset
– textContainerOrigin
– invalidateTextContainerOrigin
– layoutManager
– textStorage

Setting graphic attributes

– setBackgroundColor:
– backgroundColor
– setDrawsBackground:
– drawsBackground

Controlling display

– setNeedsDisplayInRect:avoidAdditionalLayout:
– shouldDrawInsertionPoint
– drawInsertionPointInRect:color:turnedOn:
– setConstrainedFrameSize:
– cleanUpAfterDragOperation

Setting behavioral attributes

– setEditable:
– isEditable
– setSelectable:
– isSelectable
– setFieldEditor:
– isFieldEditor
– setRichText:
– isRichText
– setImportsGraphics:
– importsGraphics

Using the Font Panel and menu

– setUsesFontPanel:
– usesFontPanel

Using the ruler

– setUsesRuler:
– usesRuler
– setRulerVisible:
– isRulerVisible

Managing the selection

– setSelectedRange:
– selectedRange
– setSelectedRange:affinity:stillSelecting:
– selectionAffinity
– setSelectionGranularity:
– selectionGranularity
– setInsertionPointColor:
– insertionPointColor
– updateInsertionPointStateAndRestartTimer:
– setSelectedTextAttributes:
– selectedTextAttributes
– markedRange
– setMarkedTextAttributes:
– markedTextAttributes

Managing the pasteboard

– preferredPasteboardTypeFromArray:restrictedToTypesFromArray:
– readSelectionFromPasteboard:
– readSelectionFromPasteboard:type:
– readablePasteboardTypes
– writablePasteboardTypes
– writeSelectionToPasteboard:type:
– writeSelectionToPasteboard:types:

Setting text attributes

– alignJustified:
– changeColor:
– setAlignment:range:
– setTypingAttributes:
– typingAttributes
– useStandardKerning:
– lowerBaseline:
– raiseBaseline:
– turnOffKerning:
– loosenKerning:
– tightenKerning:
– useStandardLigatures:
– turnOffLigatures:
– useAllLigatures:

Other action methods

– clickedOnLink:atIndex:
– pasteAsPlainText:
– pasteAsRichText:

Methods that subclasses should use or override

– updateFontPanel
– updateRuler
– acceptableDragTypes
– updateDragTypeRegistration
– selectionRangeForProposedRange:granularity:
– rangeForUserCharacterAttributeChange
– rangeForUserParagraphAttributeChange
– rangeForUserTextChange
– shouldChangeTextInRange:replacementString:
– didChangeText
– setSmartInsertDeleteEnabled:
– smartInsertDeleteEnabled
– smartDeleteRangeForProposedRange:
– smartInsertForString:replacingRange:beforeString:afterString:

Changing first responder status

– resignFirstResponder
– becomeFirstResponder
– validRequestorForSendType:returnType:

Working with the spelling checker

– spellCheckerDocumentTag

NSRulerView client methods

– rulerView:didMoveMarker:
– rulerView:didRemoveMarker:
– rulerView:didAddMarker:
– rulerView:shouldMoveMarker:
– rulerView:shouldAddMarker:
– rulerView:willMoveMarker:toLocation:
– rulerView:shouldRemoveMarker:
– rulerView:willAddMarker:atLocation:
– rulerView:handleMouseDown:

Assigning a delegate

– setDelegate:
– delegate

Class Methods

registerForServices

+ (void)**registerForServices**

Registers send and return types for the Services facility. This method is invoked automatically; you should never need to invoke it directly.

Instance Methods

acceptableDragTypes

– (NSArray *)**acceptableDragTypes**

Returns the data types that the receiver accepts as the destination view of a dragging operation. These types are automatically registered as necessary by the NSTextView. Subclasses should override this method as necessary to add their own types to those returned by NSTextView's implementation. They must then also override the appropriate methods of the NSDraggingDestination protocol to support import of those types. See that protocol's specification for more information.

See also: – **updateDragTypeRegistration**

alignJustified:

– (void)**alignJustified:**(id)*sender*

This action method applies full justification to selected paragraphs (or all text, if the receiver is a plain text object).

See also: – **alignCenter:** (NSText), – **alignLeft:** (NSText), – **alignRight:** (NSText), – **alignment** (NSText), – **setAlignment:** (NSText)

backgroundColor

– (NSColor *)**backgroundColor**

Returns the receiver's background color.

See also: – **drawsBackground**, – **setBackgroundColor:**

becomeFirstResponder

– (BOOL)**becomeFirstResponder**

Informs the receiver that it's becoming the first responder. If the previous first responder was not an NSTextView on the same NSLayoutManager as the receiving NSTextView, this method draws the selection and updates the insertion point if necessary. Returns YES.

Use NSWindow's **makeFirstResponder:**, not this method, to make an NSTextView the first responder. Never invoke this method directly.

See also: – **resignFirstResponder**

changeColor:

– (void)**changeColor:**(id)*sender*

Invoked by the NSColorPanel (*sender*) to set the color of the selected text. NSTextView's implementation queries *sender* for the color by sending it a **color** message.

cleanUpAfterDragOperation

– (void)**cleanUpAfterDragOperation**

Releases the drag information still existing after the dragging session has completed. Subclasses may override this method to clean up any additional data structures used for dragging. In your overridden method, make sure to invoke the superclass' version.

clickedOnLink:atIndex:

– (void)**clickedOnLink:**(id)*link* **atIndex:**(unsigned int)*charIndex*

Notifies the delegate that the user clicked in a *link* at the specified *charIndex*. The delegate may take any appropriate actions to handle the click in its **textView:clickedOnLink:atIndex:** method.

See also: – **textView:clickedOnLink:atIndex:** (delegate method)

delegate

– (id)**delegate**

Returns the delegate used by the receiver (and by all other NSTextViews sharing the receiver's NSLayoutManager), or **nil** if there is none.

See also: – **setDelegate:**

didChangeText

– (void)**didChangeText**

Invoked automatically at the end of a series of changes, this method posts an NSTextDidChangeNotification to the default notification center, which also results in the delegate receiving an NSText-delegate **textDidChange:** message. Subclasses implementing methods that change their text should invoke this method at the end of those methods. See the class description for more information.

See also: – **shouldChangeTextInRange:replacementString:**

dragOperationForDraggingInfo:type:

– (unsigned int)**dragOperationForDraggingInfo:**(id <NSDraggingInfo>)*dragInfo*
type:(NSString *)*type*

Returns the type of drag operation that should be performed if the image were released now. *type* is the pasteboard type that will be read from the dragging pasteboard , and *dragInfo* is an object the Application Kit creates which holds information about the dragging session. The returned value should be one of the following:

Option	Meaning
NSDragOperationCopy	The data represented by the image will be copied.
NSDragOperationLink	The data will be shared.

Option	Meaning
NSDragOperationGeneric	The operation will be defined by the destination.
NSDragOperationPrivate	The operation is negotiated privately between the source and the destination.

If none of the operations is appropriate, this method should return NSDragOperationNone.

This method is called repeatedly from **draggingEntered:** and **draggingUpdated:** as the user drags the image.

See also: **draggingEntered:** (NSDraggingDestination), **draggingUpdated:** (NSDraggingDestination)

drawInsertionPointInRect:color:turnedOn:

– (void)**drawInsertionPointInRect:**(NSRect)*aRect*
color:(NSColor *)*aColor*
turnedOn:(BOOL)*flag*

If *flag* is YES, draws the insertion point in *aRect* using *aColor*. If *flag* is NO, this method erases the insertion point. The PostScript focus must be locked on the receiver when this method is invoked.

See also: – **insertionPointColor**, – **shouldDrawInsertionPoint**, – **backgroundColor**, – **lockFocus** (NSView)

drawsBackground

– (BOOL)**drawsBackground**

Returns YES if the receiver draws its background, NO if it doesn't.

See also: – **backgroundColor**, – **setDrawsBackground:**

encodeWithCoder:

@protocol NSCoding
– (void)**encodeWithCoder:**(NSCoder *)*encoder*

Raises an NSInternalInconsistencyException. NSTextView doesn't support coding.

importsGraphics

– (BOOL)**importsGraphics**

Returns YES if the text views sharing the receiver's NSLayoutManager allow the user to import files by dragging, NO if they don't.

A text view that accepts dragged files is also a rich text view.

See also: – **isRichText**, – **textStorage**, + **attributedStringWithAttachment:** (NSAttributedString Additions), – **insertAttributedString:atIndex:** (NSMutableAttributedString), – **setImportsGraphics:**

initWithCoder:

@protocol NSCoding
– (id)**initWithCoder:**(NSCoder *)*decoder*

Raises an NSInternalInconsistencyException. NSTextView doesn't support coding.

initWithFrame:

– (id)**initWithFrame:**(NSRect)*frameRect*

Initializes a newly allocated NSTextView object with *frameRect* as its frame rectangle. This method creates the entire collection of objects associated with an NSTextView—its NSTextContainer, NSLayoutManager, and NSTextStorage—and invokes **initWithFrame:textContainer:**. Returns **self**.

This method creates the text web in such a manner that the NSTextView object is the principal owner of the objects in the web. See "The OPENSTEP Text System" for a detailed description of ownership issues.

initWithFrame:textContainer:

– (id)**initWithFrame:**(NSRect)*frameRect* **textContainer:**(NSTextContainer *)*aTextContainer*

Initializes a newly allocated NSTextView object with *frameRect* as its frame rectangle and *aTextContainer* as its text container. This method is the designated initializer for NSTextView objects. Returns **self**.

Unlike **initWithFrame:**, which builds up an entire group of text-handling objects, you use this method after you've created the other components of the text handling system—an NSTextStorage object, an NSLayoutManager object, and an NSTextContainer object. Assembling the components in this fashion means that the NSTextStorage, not the NSTextView, is the principal owner of the component objects. See "The OPENSTEP Text System" for a detailed description of ownership issues.

See also: – **initWithFrame:**

insertText:

– (void)**insertText:**(NSString *)*aString*

Inserts *aString* into the receiver's text at the insertion point if there is one, otherwise replacing the selection. The inserted text is assigned the current typing attributes, as explained in the class description under "Setting Text Attributes".

This method is the means by which typed text enters an NSTextView. See the NSInputManager class and NSTextInput protocol specifications for more information.

See also: – **typingAttributes**

insertionPointColor

– (NSColor *)**insertionPointColor**

Returns the color used to draw the insertion point.

See also: – **drawInsertionPointInRect:color:turnedOn:**, – **shouldDrawInsertionPoint**, – **setInsertionPointColor:**

invalidateTextContainerOrigin

– (void)**invalidateTextContainerOrigin**

Informs the receiver that it needs to recalculate the origin of its text container, usually because it's been resized or the contents of the text container have changed. This method is invoked automatically; you should never need to invoke it directly.

See also: – **textContainer**, – **textContainerOrigin**

isEditable

– (BOOL)**isEditable**

Returns YES if the text views sharing the receiver's NSLayoutManager allow the user to edit text, NO if they don't. If a text view is editable, it's also selectable.

See also: – **isSelectable**, – **setEditable:**

isFieldEditor

– (BOOL)**isFieldEditor**

Returns YES if the text views sharing the receiver's NSLayoutManager interpret Tab, Shift-Tab, and Return (Enter) as cues to end editing, and possibly to change the first responder; no if they accept them as text input. See the NSWindow class specification for more information on field editors. By default, NSTextViews don't behave as field editors.

See also: – **setFieldEditor:**

isRichText

– (BOOL)**isRichText**

Returns YES if the text views sharing the receiver's NSLayoutManager allow the user to apply attributes to specific ranges of the text, NO if they don't.

See also: – **importsGraphics**, – **textStorage**, – **setRichText:**

isRulerVisible

– (BOOL)**isRulerVisible**

Returns YES if the scroll view enclosing the text views sharing the receiver's NSLayoutManager shows its ruler, NO otherwise.

See also: – **usesRuler**, – **setRulerVisible:**, – **toggleRuler:** (NSText)

isSelectable

– (BOOL)**isSelectable**

Returns YES if the text views sharing the receiver's NSLayoutManager allow the user to select text, NO if they don't.

See also: – **isEditable**, – **setSelectable:**

layoutManager

– (NSLayoutManager *)**layoutManager**

Returns the NSLayoutManager that lays out text for the receiver's text container, or **nil** if there's no such object (which is the case when a text view isn't linked into a group of text objects).

See also: – **textContainer**, – **setLayoutManager:** (NSTextContainer), – **replaceLayoutManager:** (NSTextContainer)

loosenKerning:

– (void)**loosenKerning:**(id)*sender*

This action method increases the space between glyphs in the receiver's selection, or in all text if the receiver is a plain text view. Kerning values are determined by the point size of the fonts in the selection.

See also: – **tightenKerning:**, – **turnOffKerning:**, – **useStandardKerning:**

lowerBaseline:

– (void)**lowerBaseline:**(id)*sender*

This action method lowers the baseline offset of selected text by one point, or of all text if the receiver is a plain text view. As such, this method defines a more primitive operation than subscripting.

See also: – **raiseBaseline:**, – **subscript:** (NSText), – **unscript:** (NSText)

markedRange

– (NSRange)**markedRange**

Returns the range of marked text. If there's no marked text, returns a range whose location is NSNotFound.

See also: – **setMarkedTextAttributes:**

markedTextAttributes

– (NSDictionary *)**markedTextAttributes**

Returns the attributes used to draw marked text.

See also: – **setMarkedTextAttributes:**

pasteAsPlainText:

– (void)**pasteAsPlainText:**(id)*sender*

This action method inserts the contents of the pasteboard into the receiver's text as plain text, in the manner of **insertText:**.

See also: – **pasteAsRichText:**, – **insertText:**

pasteAsRichText:

– (void)**pasteAsRichText:**(id)*sender*

This action method inserts the contents of the pasteboard into the receiver's text as rich text, maintaining its attributes. The text is inserted at the insertion point if there is one, otherwise replacing the selection.

See also: – **pasteAsRichText:**, – **insertText:**

preferredPasteboardTypeFromArray:restrictedToTypesFromArray:

– (NSString *)**preferredPasteboardTypeFromArray:**(NSArray *)*availableTypes*
restrictedToTypesFromArray:(NSArray *)*allowedTypes*

Returns whatever type on the pasteboard would be most preferred for copying data. The *availableTypes* parameter lists the types that are currently available on the pasteboard. If the *allowedTypes* parameter is non-**nil** then only types in that array may be returned; otherwise, if *allowedTypes* is **nil**, any of the available pasteboard types may be returned.

You should not need to override this method. You should also not need to invoke it unless you were implementing a new type of pasteboard to handle services other than copy/paste or dragging.

See also: **– pasteAsPlainText:**, **– pasteAsRichText:**

raiseBaseline:

– (void)**raiseBaseline:**(id)*sender*

This action method raises the baseline offset of selected text by one point, or of all text if the receiver is a plain text view. As such, this method defines a more primitive operation than superscripting.

See also: – **lowerBaseline:**, – **superscript:** (NSText), – **unscript:** (NSText)

rangeForUserCharacterAttributeChange

– (NSRange)rangeForUserCharacterAttributeChange

Returns the range of characters affected by an action method that changes character (not paragraph) attributes, such as the NSText action method **changeFont:**. For rich text this is typically the range of the selection. For plain text this is the entire contents of the receiver.

If the receiver isn't editable or doesn't use the Font Panel, the range returned has a location of NSNotFound.

See also: – **rangeForUserParagraphAttributeChange**, – **rangeForUserTextChange**, – **isEditable**, – **usesFontPanel**

rangeForUserParagraphAttributeChange

– (NSRange)rangeForUserParagraphAttributeChange

Returns the range of characters affected by a method that changes paragraph (not character) attributes, such as the NSText action method **alignLeft:**. For rich text this is typically calculated by extending the range of the selection to paragraph boundaries. For plain text this is the entire contents of the receiver.

If the receiver isn't editable the range returned has a location of NSNotFound.

See also: – **rangeForUserParagraphAttributeChange**, – **rangeForUserTextChange**, – **isEditable**, – **usesRuler**

rangeForUserTextChange

– (NSRange)rangeForUserTextChange

Returns the range of characters affected by a method that changes characters (as opposed to attributes), such as **insertText:**. This is typically the range of the selection.

If the receiver isn't editable or doesn't use a ruler, the range returned has a location of NSNotFound.

See also: – **rangeForUserParagraphAttributeChange**, – **rangeForUserTextChange**, – **isEditable**, – **usesRuler**

readablePasteboardTypes

– (NSArray *)**readablePasteboardTypes**

Returns an array of strings describing the types this text view can read immediately from the pasteboard. The strings are ordered by the default preferences.

You can override this method to provide support for new types of data. If you want to add support for the default types, you can invoke the superclass version of this method or add the types directly in your overridden version.

See also: **– preferredPasteboardTypeFromArray:restrictedToTypesFromArray:**, **– writablePasteboardTypes**

readSelectionFromPasteboard:

– (BOOL)**readSelectionFromPasteboard:**(NSPasteboard *)*pboard*

Reads the text view's preferred type of data from the pasteboard specified by the *pboard* parameter. This method invokes the **preferredPasteboardTypeFromArray:restrictedToTypesFromArray:** method to determine the text view's preferred type of data and then reads the data using the **readSelectionFromPasteboard:type:** method. Returns YES if the data was successfully read.

You should not need to override this method. You might need to invoke this method if you are implementing a new type of pasteboard to handle services other than copy/paste or dragging.

See also: **– preferredPasteboardTypeFromArray:restrictedToTypesFromArray:**, **– readSelectionFromPasteboard:type:**

readSelectionFromPasteboard:type:

– (BOOL)**readSelectionFromPasteboard:**(NSPasteboard *)*pboard* **type:**(NSString *)*type*

Reads data of the given *type* from *pboard*. The new data is placed at the current insertion point, replacing the current selection if one exists. Returns YES if the data was successfully read.

You should override this method to read pasteboard types other than the default types. Use the **rangeForUserTextChange** method to obtain the the range of characters (if any) to be replaced by the new data.

See also: **– rangeForUserTextChange**

replaceTextContainer:

– (void)**replaceTextContainer:**(NSTextContainer *)*aTextContainer*

Replaces the NSTextContainer for the group of text-system objects containing the receiver with *aTextContainer*, keeping the association between the receiver and its layout manager intact, unlike **setTextContainer:**. Raises NSInvalidArgumentException if *aTextContainer* is **nil**.

See also: – **initWithFrame:textContainer:**, – **setTextContainer:**

resignFirstResponder

– (BOOL)**resignFirstResponder**

Notifies the receiver that it's been asked to relinquish its status as first responder in its NSWindow. If the object that will become the new first responder is an NSTextView attached to the same NSLayoutManager as the receiver, this method returns YES with no further action. Otherwise, this method sends a **textShouldEndEditing:** message to its delegate (if any). If the delegate returns NO, this method returns NO. If the delegate returns YES this method hides the selection highlighting and posts an NSTextDidEndEditingNotification to the default notification center.

Use NSWindow's **makeFirstResponder:**, not this method, to make an NSTextView the first responder. Never invoke this method directly.

See also: – **becomeFirstResponder**

rulerView:didAddMarker:

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

This NSRulerView client method modifies the paragraph style of the paragraphs containing the selection to accommodate a new NSTextTab represented by *aMarker*. It then records the change by invoking **didChangeText**.

NSTextView checks for permission to make the change in its **rulerView:shouldAddMarker:** method, which invokes **shouldChangeTextInRange:replacementString:** to send out the proper request and notifications, and only invokes this method if permission is granted.

See also: – **representedObject** (NSRulerMarker), – **rulerView:didMoveMarker:**, – **rulerView: didRemoveMarker:**

rulerView:didMoveMarker:

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

This NSRulerView client method modifies the paragraph style of the paragraphs containing the selection to record the new location of the NSTextTab represented by *aMarker*. It then records the change by invoking **didChangeText**.

NSTextView checks for permission to make the change in its **rulerView:shouldMoveMarker:** method, which invokes **shouldChangeTextInRange:replacementString:** to send out the proper request and notifications, and only invokes this method if permission is granted.

See also: – **representedObject** (NSRulerMarker), – **rulerView:didAddMarker:**, – **rulerView: didRemoveMarker:**

rulerView:didRemoveMarker:

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

This NSRulerView client method modifies the paragraph style of the paragraphs containing the selection—if possible—by removing the NSTextTab represented by *aMarker*. It then records the change by invoking **didChangeText**.

NSTextView checks for permission to move or remove a tab stop in its **rulerView:shouldMoveMarker:** method, which invokes **shouldChangeTextInRange:replacementString:** to send out the proper request and notifications, and only invokes this method if permission is granted.

See also: – **representedObject** (NSRulerMarker), – **shouldChangeTextInRange:replacementString:**, – **rulerView:didAddMarker:**, – **rulerView:didMoveMarker:**

rulerView:handleMouseDown:

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

This NSRulerView client method adds a left tab marker to the ruler, but a subclass can override this method to provide other behavior, such as creating guidelines. This method is invoked once with *theEvent* when the user first clicks in the *aRulerView*'s ruler area, as described in the NSRulerView class specification.

rulerView:shouldAddMarker:

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

This NSRulerView client method controls whether a new tab stop can be added. The receiver checks for permission to make the change by invoking **shouldChangeTextInRange:replacementString:** and returning the return value of that message. If the change is allowed, the receiver is then sent a **rulerView:didAddMarker:** message.

See also: – **rulerView:shouldMoveMarker:**, – **rulerView:shouldRemoveMarker:**

rulerView:shouldMoveMarker:

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

This NSRulerView client method controls whether an existing tab stop can be moved. The receiver checks for permission to make the change by invoking **shouldChangeTextInRange:replacementString:** and returning the return value of that message. If the change is allowed, the receiver is then sent a **rulerView:didMoveMarker:** message.

See also: – **rulerView:shouldAddMarker:**, – **rulerView:shouldRemoveMarker:**

rulerView:shouldRemoveMarker:

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

This NSRulerView client method controls whether an existing tab stop can be removed. Returns YES if *aMarker* represents an NSTextTab, NO otherwise. Because this method can be invoked repeatedly as the user drags a ruler marker, it returns that value immediately. If the change is allows and the user actually removes the marker, the receiver is also sent a **rulerView:didRemoveMarker:** message.

See also: – **rulerView:shouldAddMarker:**, – **rulerView:shouldMoveMarker:**

rulerView:willAddMarker:atLocation:

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

This NSRulerView client method ensures that the proposed *location* of *aMarker* lies within the appropriate bounds for the receiver's text container, returning the modified location.

See also: – **rulerView:didAddMarker:**

rulerView:willMoveMarker:toLocation:

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

This NSRulerView client method ensures that the proposed *location* of *aMarker* lies within the appropriate bounds for the receiver's text container, returning the modified location.

See also: – **rulerView:didMoveMarker:**

selectedRange

– (NSRange)**selectedRange**

Returns the range of characters selected in the receiver's layout manager.

See also: – **selectedTextAttributes**, – **setSelectedRange:affinity:stillSelecting:**,
– **selectionRangeForProposedRange:granularity:**, – **setSelectedRange:**

selectedTextAttributes

– (NSDictionary *)**selectedTextAttributes**

Returns the attributes used to indicate the selection. This is typically just the text background color.

See also: – **selectedRange**, – **setSelectedTextAttributes:**

selectionAffinity

– (NSSelectionAffinity)**selectionAffinity**

Returns the preferred direction of selection, either NSSelectionAffinityUpstream or NSSelectionAffinityDownstream. Selection affinity determines whether, for example, the insertion point appears after the last character on a line or before the first character on the following line in cases where text wraps across line boundaries.

See also: – **setSelectedRange:affinity:stillSelecting:**

selectionGranularity

– (NSSelectionGranularity)**selectionGranularity**

Returns the current selection granularity, used during mouse tracking to modify the range of the selection. This is one of:

NSSelectByCharacter
NSSelectByWord
NSSelectByParagraph

See also: – **selectionRangeForProposedRange:granularity:**, – **setSelectionGranularity:**

selectionRangeForProposedRange:granularity:

– (NSRange)**selectionRangeForProposedRange:**(NSRange)*proposedSelRange* **granularity:** (NSSelectionGranularity)*granularity*

Adjusts the *proposedSelRange* if necessary, based on *granularity*, which is one of:

NSSelectByCharacter
NSSelectByWord
NSSelectByParagraph

Returns the adjusted range. This method is invoked repeatedly during mouse tracking to modify the range of the selection. Override this method to specialize selection behavior.

See also: – **setSelectionGranularity:**

setAlignment:range:

– (void)**setAlignment:**(NSTextAlignment)*alignment* **range:**(NSRange)*aRange*

Sets the alignment of the paragraphs containing characters in *aRange* to *alignment*, which is one of:

NSLeftTextAlignment
NSRightTextAlignment
NSCenterTextAlignment
NSJustifiedTextAlignment
NSNaturalTextAlignment

See also: – **rangeForUserParagraphAttributeChange**

setBackgroundColor:

– (void)**setBackgroundColor:**(NSColor *)*aColor*

Sets the receiver's background color to *aColor*.

See also: – **setDrawsBackground:**, – **backgroundColor**

setConstrainedFrameSize:

– (void)**setConstrainedFrameSize:**(NSSize)*desiredSize*

Attempts to set the frame size for the NSTextView to *desiredSize*, constrained by the receiver's existing minimum and maximum sizes and by whether resizing is permitted.

See also: – **minSize** (NSText), – **maxSize** (NSText), – **isHorizontallyResizable** (NSText), – **isVerticallyResizable** (NSText)

setDelegate:

– (void)**setDelegate:**(id)*anObject*

Sets the delegate for all NSTextViews sharing the receiver's NSLayoutManager to *anObject*, without retaining it.

See also: – **delegate**

setDrawsBackground:

– (void)**setDrawsBackground:**(BOOL)*flag*

Controls whether the receiver draws its background. If *flag* is YES, the receiver fills its background with the background color; if *flag* is NO, it doesn't.

See also: – **setBackgroundColor:**, – **drawsBackground**

setEditable:

– (void)**setEditable:**(BOOL)*flag*

Controls whether the text views sharing the receiver's NSLayoutManager allow the user to edit text. If *flag* is YES, they allow the user to edit text and attributes; if *flag* is NO, they don't. If n NSTextView is made editable, it's also made selectable. NSTextViews are by default editable.

See also: – **setSelectable:**, – **isEditable**

setFieldEditor:

– (void)**setFieldEditor:**(BOOL)flag

Controls whether the text views sharing the receiver's NSLayoutManager interpret Tab, Shift-Tab, and Return (Enter) as cues to end editing, and possibly to change the first responder. If *flag* is YES, they do; if *flag* is NO, they don't, instead accepting these characters as text input. See the NSWindow class specification for more information on field editors. By default, NSTextViews don't behave as field editors.

See also: – **isFieldEditor**

setImportsGraphics:

– (void)**setImportsGraphics:**(BOOL)flag

Controls whether the text views sharing the receiver's NSLayoutManager allow the user to import files by dragging. If *flag* is YES, they do; if *flag* is NO, they don't. If an NSTextView is set to accept dragged files, it's also set for rich text. By default, NSTextViews don't accept dragged files.

See also: – **textStorage**, – **setRichText:**, – **importsGraphics**

setInsertionPointColor:

– (void)**setInsertionPointColor:**(NSColor *)*aColor*

Sets the color of the insertion point to *aColor.*

See also: – **drawInsertionPointInRect:color:turnedOn:**, – **shouldDrawInsertionPoint**, – **insertionPointColor**

setMarkedTextAttributes:

– (void)**setMarkedTextAttributes:**(NSDictionary *)*attributes*

Sets the attributes used to draw marked text to *attributes*. Text color, background color, and underline are the only supported attributes for marked text.

See also: – **markedTextAttributes**, – **markedRange**

setNeedsDisplayInRect:avoidAdditionalLayout:

– (void)**setNeedsDisplayInRect:**(NSRect)*aRect* **avoidAdditionalLayout:**(BOOL)*flag*

Marks the receiver as requiring display within *aRect*. If *flag* is YES, the receiver won't perform any layout that might be required to complete the display, even if this means that portions of the NSTextView remain empty. If *flag* is NO, the receiver performs at least as much layout as needed to display *aRect*.

NSTextView overrides the NSView **setNeedsDisplayInRect:** method such that it invokes this method with NO as *flag*.

setRichText:

– (void)**setRichText:**(BOOL)*flag*

Controls whether the text views sharing the receiver's NSLayoutManager allow the user to apply attributes to specific ranges of the text. If *flag* is YES they do; if *flag* is NO they don't. If *flag* is NO, they're also set not to accept dragged files. By default, NSTextViews let the user apply multiple attributes to text, but don't accept dragged files.

See also: – **textStorage**, – **isRichText**, – **setImportsGraphics:**

setRulerVisible:

– (void)**setRulerVisible:**(BOOL)*flag*

Controls whether the scroll view enclosing text views sharing the receiver's NSLayoutManager displays the ruler. If *flag* is YES it shows the ruler; if *flag* is NO it hides the ruler. By default, the ruler is not visible.

See also: – **setUsesRuler:**, – **isRulerVisible**, – **toggleRuler:** (NSText)

setSelectable:

– (void)**setSelectable:**(BOOL)*flag*

Controls whether the text views sharing the receiver's NSLayoutManager allow the user to select text. If *flag* is YES, they do; if *flag* is NO, they don't. If an NSTextView is made not selectable, it's also made not editable. NSTextViews are by default both editable and selectable.

See also: – **setEditable:**, – **isSelectable**

setSelectedRange:

– (void)**setSelectedRange:**(NSRange)*charRange*

Sets the selection to the characters in *charRange*, resets the selection granularity to NSSelectByCharacter, posts an NSTextViewDidChangeSelectionNotification to the default notification center. Also removes the marking from marked text if the new selection is greater than the marked region.

charRange must begin and end on glyph boundaries and not split base glyphs and their non-spacing marks.

See also: – **setSelectedRange:affinity:stillSelecting:**, – **selectionAffinity**, – **selectionGranularity**, – **selectedRange**

setSelectedRange:affinity:stillSelecting:

– (void)**setSelectedRange:**(NSRange)*charRange*
affinity:(NSSelectionAffinity)*affinity*
stillSelecting:(BOOL)*flag*

Sets the selection to the characters in *charRange*, using *affinity* if needed to determine how to display the selection or insertion point (see the description for **selectionAffinity** for more information). *flag* indicates whether this method is being invoked during mouse-dragging or after the user releases the mouse. If *flag* is YES the receiver doesn't send notifications or remove the marking from its marked text; if *flag* is NO it does as appropriate. This method also resets the selection granularity to NSSelectByCharacter.

charRange must begin and end on glyph boundaries and not split base glyphs and their non-spacing marks.

See also: – **setSelectedRange:**, – **selectionAffinity**, – **selectionGranularity**, – **setSelectedRange:**

setSelectedTextAttributes:

– (void)setSelectedTextAttributes:(NSDictionary *)*attributes*

Sets the attributes used to indicate the selection to *attributes*. Text color, background color, and underline are the only supported attributes for selected text.

See also: – **selectedRange**, – **selectedTextAttributes**

setSelectionGranularity:

– (void)**setSelectionGranularity:**(NSSelectionGranularity)*granularity*

Sets the selection granularity for subsequent extension of a selection to *granularity*, which may be one of:

NSSelectByCharacter
NSSelectByWord
NSSelectByParagraph

Selection granularity is used to determine how the selection is modified when the user Shift-clicks or drags the mouse after a double- or triple-click. For example, if the user selects a word by double-clicking, the selection granularity is set to NSSelectByWord. Subsequent shift-clicks then extend the selection by words.

Selection granularity is reset to NSSelectByCharacter whenever the selection is set. You should always set the selection granularity after setting the selection.

See also: – **selectionGranularity**, – **setSelectedRange:**

setSmartInsertDeleteEnabled:

– (void)setSmartInsertDeleteEnabled:(BOOL)*flag*

Controls whether the receiver inserts or deletes space around selected words so as to preserve proper spacing and punctuation. If *flag* is YES it does; if *flag* is NO it inserts and deletes exactly what's selected.

See also: – **smartInsertForString:replacingRange:beforeString:afterString:**,
– **smartDeleteRangeForProposedRange:**, – **smartInsertDeleteEnabled**

setTextContainer:

– (void)**setTextContainer:**(NSTextContainer *)*aTextContainer*

Sets the receiver's text container to *aTextContainer*. The receiver then uses the layout manager and text storage of *aTextContainer*. This method is invoked automatically when you create an NSTextView; you should never invoke it directly, but might want to override it. To change the text view for an established group of text-system objects, send **setTextView:** to the text container. To replace the text container for a text

view and maintain the view's association with the existing layout manager and text storage, use **replaceTextContainer:**.

See also: – **textContainer**

setTextContainerInset:

– (void)**setTextContainerInset:**(NSSize)*inset*

Sets the empty space the NSTextView leaves around its associated text container to *inset*.

See also: – **textContainerOrigin**, – **invalidateTextContainerOrigin**, – **textContainerInset**

setTypingAttributes:

– (void)**setTypingAttributes:**(NSDictionary *)*attributes*

Sets the receiver's typing attributes to *attributes*. Typing attributes are reset automatically whenever the selection changes. If you add any user actions that change text attributes, you should use this method to apply those attributes to a zero-length selection.

See also: – **typingAttributes**

setUsesFontPanel:

– (void)**setUsesFontPanel:**(BOOL)*flag*

Controls whether the text views sharing the receiver's NSLayoutManager use the Font Panel and Font menu. If *flag* is YES, they respond to messages from the Font Panel and from the Font menu, and update the Font Panel with the selection font whenever it changes. If *flag* is NO they disallow character attribute changes. By default, NSTextView objects use the Font Panel and menu.

See also: – **rangeForUserCharacterAttributeChange**, – **usesFontPanel**

setUsesRuler:

– (void)**setUsesRuler:**(BOOL)*flag*

Controls whether the text views sharing the receiver's NSLayoutManager use an NSRulerView and respond to Format menu commands. If *flag* is YES, they respond to NSRulerView client messages and to paragraph-related menu actions, and update the ruler (when visible) as the selection changes with its paragraph and tab attributes. If *flag* is NO, the ruler is hidden and the text views disallow paragraph attribute changes. By default, NSTextView objects use the ruler.

See also: – **setRulerVisible:**, – **rangeForUserParagraphAttributeChange**, – **usesRuler**

shouldChangeTextInRange:replacementString:

– (BOOL)**shouldChangeTextInRange:**(NSRange)*affectedCharRange* **replacementString:** (NSString *)*replacementString*

Initiates a series of delegate messages (and general notifications) to determine whether modifications can be made to the receiver's text. If characters in the text string are being changed, *replacementString* contains the characters that will replace the characters in *affectedCharRange*. If only text attributes are being changed, *replacementString* is **nil**. This method checks with the delegate as needed using **textShouldBeginEditing:** and **textView:shouldChangeTextInRange:replacementString:**, returning YES to allow the change, and NO to prohibit it.

This method must be invoked at the start of any sequence of user-initiated editing changes. If your subclass of NSTextView implements new methods that modify the text, make sure to invoke this method to determine whether the change should be made. If the change is allowed, complete the change by invoking the **didChangeText** method. See "Notifying About Changes to the Text" in the class description for more information. If you can't determine the affected range or replacement string before beginning changes, pass (NSNotFound, 0) and **nil** for these values.

See also: – **isEditable**

shouldDrawInsertionPoint

– (BOOL)**shouldDrawInsertionPoint**

Returns YES if the receiver should draw its insertion point, NO if the insertion point can't or shouldn't be drawn (for example, if the receiver's window isn't key).

See also: – **drawInsertionPointInRect:color:turnedOn:**

smartDeleteRangeForProposedRange:

– (NSRange)smartDeleteRangeForProposedRange:(NSRange)*proposedCharRange*

Given *proposedCharRange*, returns an extended range that includes adjacent whitespace that should be deleted along with the proposed range in order to preserve proper spacing and punctuation of the text surrounding the deletion.

NSTextView uses this method as necessary; you can also use it in implementing your own methods that delete text, typically when the selection granularity is NSSelectByWord. To do so, invoke this method with the proposed range to delete, then actually delete the range returned. If placing text on the pasteboard, however, you should put only the characters from the proposed range onto the pasteboard.

See also: – **smartInsertForString:replacingRange:beforeString:afterString:**, – **selectionGranularity**, – **smartInsertDeleteEnabled**

smartInsertDeleteEnabled

– (BOOL)smartInsertDeleteEnabled

Returns YES if the receiver inserts or deletes space around selected words so as to preserve proper spacing and punctuation, NO if it inserts and deletes exactly what's selected.

See also: – **smartInsertForString:replacingRange:beforeString:afterString:**, – **smartDeleteRangeForProposedRange:**, – **setSmartInsertDeleteEnabled:**

smartInsertAfterStringForString:replacingRange:

– (NSString *)**smartInsertAfterStringForString:**(NSString *)*aString*
replacingRange:(NSRange)*charRange*

Returns any whitespace that needs to be added after *aString* to preserve proper spacing and punctuation when *aString* is inserted into the receiver's text over *charRange*. If *aString* is **nil** or if smart insertion and deletion is disabled, this method returns **nil**.

If you're writing an Objective-C method, don't invoke this method directly. Instead, use **smartInsertForString:replacingRange:beforeString:afterString:**, which calls this method as part of its implementation.

If you're writing a Java method that inserts text, insert the following over *charRange*:

- The results of the Java version of **smartInsertBeforeStringForString:replacingRange:**
- *aString*
- The results of the Java version of this method

smartInsertBeforeStringForString:replacingRange:

– (NSString *)**smartInsertBeforeStringForString:**(NSString *)*aString*
replacingRange:(NSRange)*charRange*

Returns any whitespace that needs to be added before *aString* to preserve proper spacing and punctuation when *aString* is inserted into the receiver's text over *charRange*. If *aString* is **nil** or if smart insertion and deletion is disabled, this method returns **nil**.

If you're writing an Objective-C method, don't invoke this method directly. Instead, use **smartInsertForString:replacingRange:beforeString:afterString:**, which calls this method as part of its implementation.

If you're writing a Java method that inserts text, insert the following over *charRange*:

- The results of the Java version of this method
- *aString*
- The results of the Java version of **smartInsertAfterStringForString:replacingRange**

smartInsertForString:replacingRange:beforeString:afterString:

– **(void)smartInsertForString:(NSString *)***aString*
replacingRange:(NSRange)*charRange*
beforeString:(NSString **)*beforeString*
afterString:(NSString **)*afterString*

Determines whether whitespace needs to be added around *aString* to preserve proper spacing and punctuation when it's inserted into the receiver's text over *charRange*. Returns by reference in *beforeString* and *afterString* any whitespace that should be added, unless either or both is NULL. Both are returned as **nil** if *aString* is **nil** or if smart insertion and deletion is disabled.

As part of its implementation, this method calls **smartInsertAfterStringForString:replacingRange:** and **smartInsertBeforeStringForString:replacingRange:**.To change this method's behavior, override those two methods instead of this one.

NSTextView uses this method as necessary. You can also use it in implementing your own methods that insert text. To do so, invoke this method with the proper arguments, then insert *beforeString*, *aString*, and *afterString* in order over *charRange*.

See also: – **smartDeleteRangeForProposedRange:**, – **smartInsertDeleteEnabled**

spellCheckerDocumentTag

– (int)**spellCheckerDocumentTag**

Returns a tag identifying the NSTextView text as a document for the spell checker server. The document tag is obtained by sending a **uniqueSpellDocumentTag** message to the spell server the first time this method is invoked for a particular group of NSTextViews. See the NSSpellChecking and NSSpellServer class specifications for more information on how this tag is used.

textContainer

– (NSTextContainer *)**textContainer**

Returns the receiver's text container.

See also: – **setTextContainer:**

textContainerInset

– (NSSize)**textContainerInset**

Returns the empty space the NSTextView leaves around its text container.

See also: – **textContainerOrigin**, – **invalidateTextContainerOrigin**, – **setTextContainerInset:**

textContainerOrigin

– (NSPoint)**textContainerOrigin**

Returns the origin of the receiver's text container, which is calculated from the receiver's bounds rectangle, container inset, and the container's used rect.

See also: – **invalidateTextContainerOrigin**, – **textContainerInset**, – **usedRectForTextContainer:** (NSLayoutManager)

textStorage

– (NSTextStorage *)**textStorage**

Returns the receiver's text storage object.

tightenKerning:

– (void)**tightenKerning:**(id)*sender*

This action method decreases the space between glyphs in the receiver's selection, or for all glyphs if the receiver is a plain text view. Kerning values are determined by the point size of the fonts in the selection.

See also: – **loosenKerning:**, – **useStandardKerning:**, – **turnOffKerning:**

turnOffKerning:

– (void)**turnOffKerning:**(id)*sender*

This action method causes the receiver to use nominal glyph spacing for the glyphs in its selection, or for all glyphs if the receiver is a plain text view.

See also: – **useStandardKerning:**, – **loosenKerning:**, – **tightenKerning:**, – **isRichText**

turnOffLigatures:

– (void)**turnOffLigatures:**(id)*sender*

This action method causes the receiver to use only required ligatures when setting text, for the glyphs in the selection if the receiver is a rich text view, or for all glyphs if it's a plain text view.

See also: – **useAllLigatures:**, – **isRichText**, – **useStandardLigatures:**

typingAttributes

– (NSDictionary *)**typingAttributes**

Returns the current typing attributes.

See also: – **setTypingAttributes:**

updateDragTypeRegistration

– (void)**updateDragTypeRegistration**

If the receiver is editable and is a rich text view, causes all NSTextViews associated with the receiver's NSLayoutManager to register their acceptable drag types. If the NSTextView isn't editable or isn't rich text, causes those NSTextViews to unregister their dragged types.

Subclasses can override this method to change the conditions for registering and unregistering drag types, whether as a group or individually based on the current state of the NSTextView. They can then invoke this method when that state changes to perform that reregistration.

See also: – **acceptableDragTypes**, – **registerForDraggedTypes:** (NSView),
– **unregisterDraggedTypes** (NSView), – **isEditable**, – **importsGraphics**, – **isRichText**

updateFontPanel

– (void)**updateFontPanel**

Updates the Font Panel to contain the font attributes of the selection. Does nothing if the receiver doesn't use the Font Panel. You should never need to invoke this method directly, but you can override it if needed to handle additional font attributes.

See also: – **usesFontPanel**

updateInsertionPointStateAndRestartTimer:

– (void)**updateInsertionPointStateAndRestartTimer:**(BOOL)*flag*

Updates the insertion point's location and, if *flag* is YES, restarts the blinking cursor timer. This method is invoked automatically whenever the insertion point needs to be moved; you should never need to invoke it directly, but you can override it to add different insertion point behavior.

See also: – **shouldDrawInsertionPoint**, – **drawInsertionPointInRect:color:turnedOn:**

updateRuler

– (void)**updateRuler**

Updates the NSRulerView in the receiver's enclosing scroll view to reflect the selection's paragraph and marker attributes. Does nothing if the ruler isn't visible or if the receiver doesn't use the ruler. You should never need to invoke this method directly, but you can override this method if needed to handle additional ruler attributes.

See also: – **usesRuler**

useAllLigatures:

– (void)**useAllLigatures:**(id)*sender*

This action method causes the receiver to use all ligatures available for the fonts and languages used when setting text, for the glyphs in the selection if the receiver is a rich text view, or for all glyphs if it's a plain text view.

See also: – **turnOffLigatures:**, – **useStandardLigatures:**

usesFontPanel

– (BOOL)**usesFontPanel**

Returns YES if the text views sharing the receiver's NSLayoutManager use the Font Panel, NO otherwise. See **setUsesFontPanel:** and **rangeForUserCharacterAttributeChange** for the effect this has on an NSTextView's behavior.

usesRuler

– (BOOL)**usesRuler**

Returns YES if the text views sharing the receiver's NSLayoutManager use a ruler view, NO otherwise. See **setUsesRuler:** and **rangeForUserParagraphAttributeChange** for the effect this has on an NSTextView's behavior

See also: – **setUsesRuler:**

useStandardKerning:

– (void)**useStandardKerning:**(id)*sender*

This action method causes the receiver to use pair kerning data for the glyphs in its selection, or for all glyphs if the receiver is a plain text view. This data is taken from a font's AFM file

See also: – **isRichText**, – **loosenKerning:**, – **tightenKerning:**, – **turnOffKerning:**

useStandardLigatures:

– (void)**useStandardLigatures:**(id)*sender*

This action method causes the receiver to use the standard ligatures available for the fonts and languages used when setting text, for the glyphs in the selection if the receiver is a rich text view, or for all glyphs if it's a plain text view.

See also: – **turnOffLigatures:**, – **useAllLigatures:**

validRequestorForSendType:returnType:

– (id)**validRequestorForSendType:**(NSString *)*sendType* **returnType:**(NSString *)*returnType*

Returns **self** if *sendType* specifies a type of data the text view can put on the pasteboard and *returnType* contains a type of data the text view can read from the pasteboard, otherwise returns **nil**.

See also: **– validRequestorForSendType:returnType:** (NSResponder)

writablePasteboardTypes

– (NSArray *)**writablePasteboardTypes**

If the text view contains some selected data, this method returns an array of strings describing the types that can be written to the pasteboard immediately. You can override this method to add new supported types to the array of strings.

See also: **– readablePasteboardTypes**

writeSelectionToPasteboard:type:

– (BOOL)**writeSelectionToPasteboard:**(NSPasteboard *)*pboard* **type:**(NSString *)*type*

Writes the current selection to *pboard* using the given *type*. Returns YES if the data was successfully written. You can override this method to add support for writing new types of data to the pasteboard. You

should invoke the superclass' version of the method to handle any types of data your overridden version does not.

See also: **– readSelectionFromPasteboard:type:**

writeSelectionToPasteboard:types:

– (BOOL)**writeSelectionToPasteboard:**(NSPasteboard *)*pboard* **types:**(NSArray *)*types*

Writes the current selection to *pboard* under each type in the *types* array. Returns YES if the data for any single type was written successfully.

You should not need to override this method. You might need to invoke this method if you are implementing a new type of pasteboard to handle services other than copy/paste or dragging.

See also: **– writeSelectionToPasteboard:type:**

Methods Implemented By the Delegate

NSTextView communicates with its delegate through methods declared both by NSTextView and by its superclass, NSText. See the NSText class specification for those other delegate methods.

textView:clickedOnCell:inRect:

– (void)**textView:**(NSTextView *)*aTextView*
clickedOnCell:(id <NSTextAttachmentCell>)*attachmentCell*
inRect:(NSRect)*cellFrame*

Invoked after the user clicks on *attachmentCell* within *cellFrame* in an NSTextView and the cell wants to track the mouse. The delegate can use this message as its cue to perform an action or select the attachment cell's character. *aTextView* is the first NSTextView in a series shared by an NSLayoutManager, not necessarily the one that draws *attachmentCell*.

The delegate may subsequently receive a **textView:doubleClickedOnCell:** message if the user continues to perform a double click.

Implementing the **textView:clickedOnCell:inRect:atIndex:** method is preferred to implementing this method. If you implement that method, you do not need to implement this one.

See also: – **wantsToTrackMouse** (NSTextAttachmentCell)

textView:clickedOnCell:inRect:atIndex:

- (void)**textView:**(NSTextView *)*textView*
  clickedOnCell:(id <NSTextAttachmentCell>)*cell*
  inRect:(NSRect)*cellFrame*
  atIndex:(unsigned)*charIndex*

Invoked after the user clicks on *cell* within *cellFrame* at the specified *charIndex* in an NSTextView and the cell wants to track the mouse. The delegate can use this message as its cue to perform an action or select the attachment cell's character. *textView* is the first NSTextView in a series shared by an NSLayoutManager, not necessarily the one that draws *cell*.

The delegate may subsequently receive a **textView:doubleClickedOnCell:atIndex:** message if the user continues to perform a double click.

See also: **– textView:doubleClickedOnCell:atIndex:**

textView:clickedOnLink:

- (BOOL)**textView:**(NSTextView *)*textView* **clickedOnLink:**(id)*link*

Invoked after the user clicks on *link* in an NSTextView if the delegate does not respond to the **textView: clickedOnLink:atIndex:** message. The delegate can use this method to handle the click on the *link*.

Implementing the **textView:clickedOnLink:atIndex:** method is preferred to implementing this method. If you implement that method, you do not need to implement this one.

See also: **– clickedOnLink:atIndex:** (NSTextView), **– textView:clickedOnLink:atIndex:**

textView:clickedOnLink:atIndex:

- (BOOL)**textView:**(NSTextView *)*textView*
  clickedOnLink:(id)*link*
  atIndex:(unsigned)*charIndex*

Invoked after the user clicks on *link* at the specified *charIndex* in an NSTextView. The delegate can use this method to handle the click on the *link*. Return YES to indicate that the click was handled, otherwise return NO to allow the next responder to handle it.

See also: **– clickedOnLink:atIndex:** (NSTextView)

textView:doCommandBySelector:

– (BOOL)**textView:**(NSTextView *)*aTextView*
doCommandBySelector:(SEL)*aSelector*

Sent from NSTextView's **doCommandBySelector:**, this method allows the delegate to perform the command for the text view. If the delegate returns YES, the text view doesn't perform *aSelector*; if the delegate returns NO, the text view attempts to perform it. *aTextView* is the first NSTextView in a series shared by an NSLayoutManager.

textView:doubleClickedOnCell:inRect:

– (void)**textView:**(NSTextView *)*aTextView*
doubleClickedOnCell:(id <NSTextAttachmentCell>)*attachmentCell*
inRect:(NSRect)*cellFrame*

Invoked when the user double-clicks on *attachmentCell* within *cellFrame* in an NSTextView and the cell wants to track the mouse. The delegate can use this message as its cue to perform an action, such as opening the file represented by the attachment. *aTextView* is the first NSTextView in a series shared by an NSLayoutManager, not necessarily the one that draws *attachmentCell*.

Implementing the **textView:doubleClickedOnCell:inRect:atIndex:** method is preferred to implementing this method. If you implement that method, you do not need to implement this one.

See also: – **wantsToTrackMouse** (NSTextAttachmentCell), **– textView:doubleClickedOnCell:inRect:atIndex:**

textView:doubleClickedOnCell:inRect:atIndex:

- (void)**textView:**(NSTextView *)*textView*
doubleClickedOnCell:(id <NSTextAttachmentCell>)*cell*
inRect:(NSRect)*cellFrame*
atIndex:(unsigned)*charIndex*

Invoked when the user double-clicks on *attachmentCell* within *cellFrame* at the specified *charIndex* in an NSTextView and the cell wants to track the mouse. The delegate can use this message as its cue to perform an action, such as opening the file represented by the attachment. *aTextView* is the first NSTextView in a series shared by an NSLayoutManager, not necessarily the one that draws *attachmentCell*.

See also: – **wantsToTrackMouse** (NSTextAttachmentCell)

textView:draggedCell:inRect:event:

– (void)**textView:**(NSTextView *)*aTextView*
draggedCell:(id <NSTextAttachmentCell>)*attachmentCell*
inRect:(NSRect)*aRect*
event:(NSEvent *)*theEvent*

Invoked when the user attempts to drag *attachmentCell* from *aRect* within an NSTextView and the cell wants to track the mouse. *theEvent* is the mouse-down event that preceded the mouse-dragged event. The delegate can use this message as its cue to initiate a dragging operation.

Implementing the **textView:draggedCell:inRect:event:atIndex:** method is preferred to implementing this method. If you implement that method, you do not need to implement this one.

See also: – **wantsToTrackMouse** (NSTextAttachmentCell), – **dragImage:at:offset:event:pasteboard:source:slideBack:** (NSView), – **dragFile:fromRect:slideBack:event:** (NSView)

textView:draggedCell:inRect:event:atIndex:

- (void)**textView:**(NSTextView *)*view*
draggedCell:(id <NSTextAttachmentCell>)*cell*
inRect:(NSRect)*rect*
event:(NSEvent *)*event*
atIndex:(unsigned int)*charIndex*

Invoked when the user attempts to drag *attachmentCell* from *aRect* within an NSTextView and the cell wants to track the mouse. The *event* parameter is the mouse-down event that preceded the mouse-dragged event. The *charIndex* parameter indicates the character position where the mouse was clicked. The delegate can use this message as its cue to initiate a dragging operation.

See also: – **wantsToTrackMouse** (NSTextAttachmentCell), – **dragImage:at:offset:event:pasteboard:source:slideBack:** (NSView), – **dragFile:fromRect:slideBack:event:** (NSView)

textView:shouldChangeTextInRange:replacementString:

– (BOOL)**textView:**(NSTextView *)*aTextView*
shouldChangeTextInRange:(NSRange)*affectedCharRange*
replacementString:(NSString *)*replacementString*

Invoked when an NSTextView needs to determine if text in the range *affectedCharRange* should be changed. If characters in the text string are being changed, *replacementString* contains the characters that will replace the characters in *affectedCharRange*. If only text attributes are being changed, *replacementString* is **nil**. The delegate can return YES to allow the replacement, or NO to reject the change.

aTextView is the first NSTextView in a series shared by an NSLayoutManager.

textView:willChangeSelectionFromCharacterRange:toCharacterRange:

– (NSRange)**textView:**(NSTextView *)*aTextView*
willChangeSelectionFromCharacterRange:(NSRange)*oldSelectedCharRange*
toCharacterRange:(NSRange)*newSelectedCharRange*

Invoked before an NSTextView finishes changing the selection—that is, when the last argument to a **setSelectedRange:affinity:stillSelecting:** message is NO. *oldSelectedCharRange* is the original range of the selection. *newSelectedCharRange* is the proposed character range for the new selection. The delegate can return and adjusted range or return *newSelectedCharRange* unmodified.

aTextView is the first NSTextView in a series shared by an NSLayoutManager.

textViewDidChangeSelection:

– (void)**textViewDidChangeSelection:**(NSNotification *)*aNotification*

Invoked when the selection changes in the NSTextView. The name of *aNotification* is NSTextViewDidChangeSelectionNotification.

See also: **NSTextViewDidChangeSelectionNotification** (notification)

Notifications

NSTextView posts the following notifications as well as those declared by its superclasses, particularly NSText. See the NSText class specification for those other notifications.

NSTextViewDidChangeSelectionNotification

Posted when the selected range of characters changes. NSTextView posts this notification whenever **setSelectedRange:affinity:stillSelecting:** is invoked either directly, or through the many methods (**mouseDown:**, **selectAll:**, and so on) that invoke it indirectly. When the user is selecting text, this notification is posted only once, at the end of the selection operation. The NSTextView's delegate receives a **textViewDidChangeSelection:** message when this notification is posted.

This notification contains a notification object and a userInfo dictionary. The notification object is the notifying NSTextView. The userInfo dictionary contains these keyes and values:

Key	Value
NSOldSelectedCharacterRange	An NSValue object containing an NSRange

NSTextViewWillChangeNotifyingTextViewNotification

Posted when a new NSTextView is established as the NSTextView that sends notifications. This allows observers to reregister themselves for the new NSTextView. Methods such as **removeTextContainerAtIndex:**, **textContainerChangedTextView**:, and **insertTextContainer:atIndex:** cause this notification to be posted.

This notification contains a notification object and a userInfo dictionary. The notification object is the old notifying NSTextView, or **nil**. The userInfo dictionary contains these keyes and values:

Key	Value
NSOldNotifyingTextView	The old NSTextView, if one exists
NSNewNotifyingTextView	The new NSTextView, if one exists

There's no delegate method associated with this notification. The text-handling system ensures that when a new NSTextView replaces an old one as the notifying NSTextView, the existing delegate becomes the delegate of the new NSTextView and the delegate is registered to receive NSTextView notifications from the new notifying NSTextView. All other observers are responsible for registering themselves on receiving this notification.

See also: – **removeObserver:** (NSNotificationCenter), – **addObserver:selector:name:object:** (NSNotificationCenter)

NSView

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

Class at a Glance

Purpose

NSView is an abstract class that defines the basic drawing, event-handling, and printing architecture of an OPENSTEP application. You typically don't interact with NSView API directly; rather, your custom view classes

inherit from NSView and override many of its methods, which are invoked automatically by the Application Kit. If you're not creating a custom view class, there are few methods you need to use.

Principal Attributes

- Event handling
- Integrated display to screen and printer
- Flexible coordinate systems
- Icon dragging

Creation

Interface Builder

– initWithFrame: Designated initializer.

Commonly Used Methods

– frame	**Returns the NSView's location and size.**
– bounds	Returns the NSView's internal origin and size.
– setNeedsDisplay:	Marks the NSView as needing to be redrawn.
– window	Returns the NSWindow that contains the NSView.
– drawRect:	Draws the NSView. (All subclasses must implement this method, but it's rarely invoked explicitly.)

Class Description

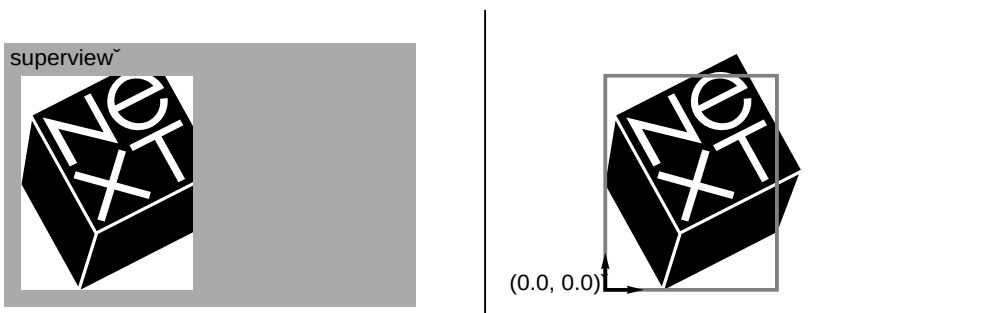
NSView is an abstract class that provides concrete subclasses with a structure for drawing, printing, and handling events. NSViews are arranged within an NSWindow, in a nested hierarchy of subviews. A view object claims a rectangular region of its enclosing superview, is responsible for all drawing within that region, and is eligible to receive mouse events occurring in it as well. In addition to these major responsibilities, NSView handles dragging of icons and works with the NSScrollView class to support efficient scrolling. The following sections explore these areas and more.

Most of the functionality of NSView is either automatically invoked by the Application Kit, or is available in Interface Builder. Unless you're implementing a concrete subclass of NSView or working intimately with the content of the view hierarchy at run time, you don't need to know much about this class's interface. See "Commonly Used Methods" above for methods you might use regardless.

The View Hierarchy

To be displayed, an NSView must be placed in an NSWindow. All view objects within an NSWindow are arranged in a hierarchy that begins at the NSWindow's *content view*, with each NSView having a single *superview* and zero or more *subviews* (see the NSWindow class specification for more on the content view). An NSView's superview and all the NSViews above the superview are sometimes referred to as the NSView's *ancestors*. An NSView's subviews and all of their subviews on down are known as the NSView's *descendants*. Each NSView in the view hierarchy has its own area to draw in and its own coordinate system, expressed as a transformation of its superview's coordinate system. An NSView can scale, translate, or rotate its coordinates dynamically, and a subclass can declare its *y* axis flipped to allow drawing from top to bottom—useful for drawing text, for example.

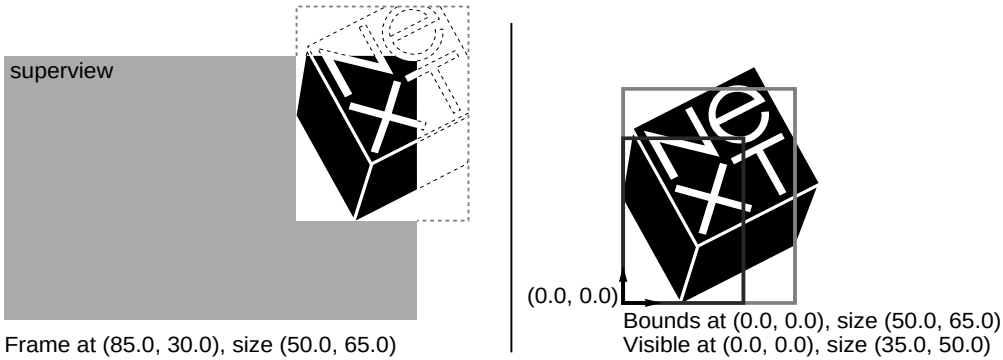
Graphically, an NSView can be regarded as a framed canvas. The frame locates the NSView in its superview, defines its size, and clips drawing to its edges, while the canvas defines the NSView's own internal coordinate system and hosts the actual drawing. The frame can be moved around, resized, and rotated in the superview, so that the NSView's image moves with it. Similarly, the canvas can be shifted, stretched, and rotated, so that the drawn image moves within the frame. The frame maps onto a region of the canvas that defines the bounds of what can possibly be seen. An NSView therefore keeps track of its space using two rectangles, one for each perspective: The *frame rectangle* gives the exterior perspective and the *bounds rectangle* give the interior. The **frame** and **bounds** methods, respectively, return these rectangles. This figure shows the relation between the frame rectangle, on the left, and the bounds rectangle over the canvas, on the right:



Frame at (5.0, 5.0), size (50.0, 65.0)˘

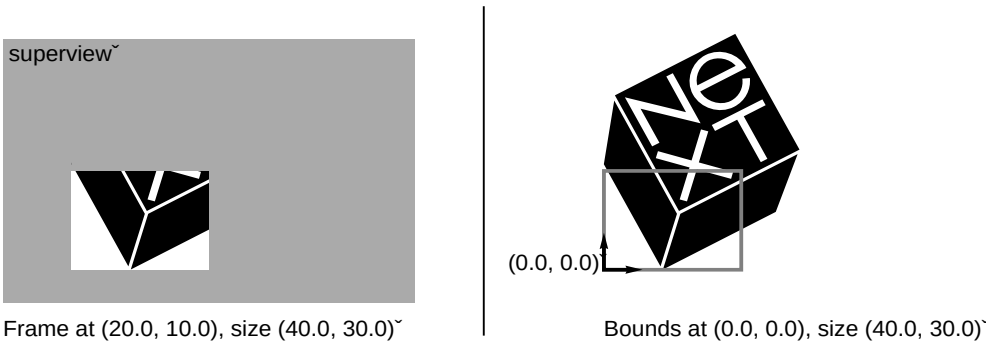
Bounds at (0.0, 0.0), size (50.0, 65.0)`

Although the bounds rectangle indicates the portion of the NSView that is potentially visible through the frame, if the frame runs outside of the superview the image will be clipped even within the bounds rectangle. An NSView's *visible rectangle* reflects the portion of an NSView that actually displays, in terms of its own coordinate system (the darker gray rectangle in the figure below). It isn't often important to know what the visible rectangle is, since the display mechanism automatically limits drawing to visible portions of a view. If a subclass must perform expensive precalculation to build its image, though, it can use the **visibleRect** method to limit its work to what's actually needed.



The **initWithFrame:** method establishes an NSView's frame rectangle, but doesn't insert it into an NSWindow's view hierarchy. This is the job of the **addSubview:** method, which you send to the NSView that you want to contain the newly initialized one. The frame rectangle is then interpreted in terms of the superview, properly locating the new NSView both by its place in the view hierarchy and its location in the superview's NSWindow.

After initialization, you can move an NSView programmatically using any of the frame-setting methods: **setFrame:**, **setFrameOrigin:**, **setFrameSize:**, and **setFrameRotation:**. When you move an NSView all of its subviews move along with it. When you change the frame rectangle's size, the bounds rectangle is automatically resized to match (see figure below), and the subviews are automatically resized as described in "Moving and Resizing NSViews" below. **setFrameRotation:** rotates the NSView around the origin of the frame rectangle (which is typically the lower left corner).



A number of methods access the view hierarchy itself. **superview** returns the receiver's containing NSView, while **subviews** returns an NSArray containing its immediate descendant NSViews. The **window** method returns the NSWindow whose view hierarchy the receiver belongs to. You can add NSViews to and remove them from the view hierarchy using the methods **addSubview:**, **removeFromSuperview**, and **replaceSubview:with:**. An additional method, **addSubview:positioned:relativeTo:**, allows you to specify the ordering of NSViews that may overlap (though laying out NSViews so that they overlap isn't recommended).

When you add a subview with **addSubview:**, the receiver retains the view. When you remove a subview from a view hierarchy with **removeFromSuperview**, or replace it with **replaceSubview:with:**, the view is released. If you want to keep using a view after removing it from a view hierarchy (if, for example, you are swapping through a number of views), you must retain it before removing or replacing it.

When an NSView is added as a subview of another view, it automatically invokes the **viewWillMoveToSuperview:** and **viewWillMoveToWindow:** methods. Concrete subclasses can override these methods, allowing an instance to query its new superview or NSWindow about relevant state and update itself accordingly. A few other methods allow you to inspect relationships among NSViews: **isDescendantOf:** confirms the containment of the receiver, **ancestorSharedWithView:** find the common container of two NSViews, and **opaqueAncestor** returns the closest containing NSView that's guaranteed to draw every pixel in the receiver's frame (possible the receiver itself).

Coordinate Conversion in the View Hierarchy

At various times, particularly when handling events, you need to convert a rectangle or point from the coordinate system of one NSView to another (typically a superview or subview). NSView defines six methods that convert rectangles, points, and sizes in either direction:

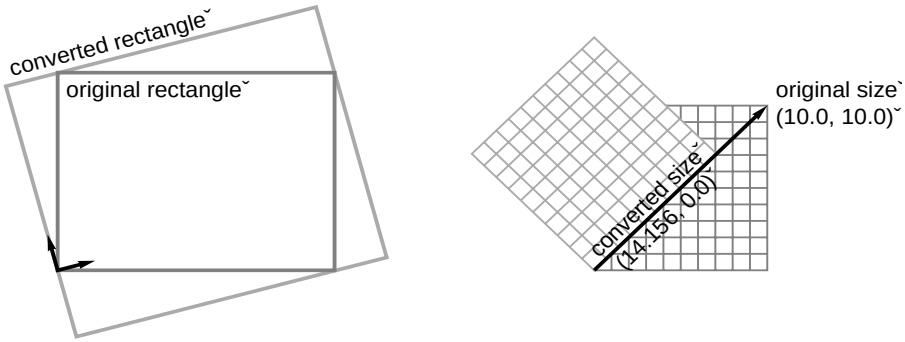
– convertPoint:fromView:	– convertPoint:toView:
– convertSize:fromView:	– convertSize:toView:
– convertRect:fromView:	– convertRect:toView:

These methods convert geometric structures between the receiver's coordinate system and another NSView's within the same NSWindow, returning an alternate expression for the same on-screen location or area. Note that the structure in question needn't actually be located within the NSView's bounds rectangle; it's merely assumed to be expressed in that NSView's coordinate system. If the second argument to a conversion method is **nil**, the conversion is made between the receiver's coordinate system and the base coordinate system of its NSWindow.

For converting to and from the screen coordinate system, NSWindow defines the **convertBaseToScreen:** and **convertScreenToBase:** methods. Using the NSView conversion methods along with these allows you to convert a geometric structure between an NSView's coordinate system and the screen's with only two messages.

Conversion is straightforward when neither NSView is rotated, or when dealing only with points. When converting rectangles or sizes between NSViews with different rotations, the geometric structure must be altered in a reasonable way. In converting a rectangle NSView makes the assumption that you want to guarantee coverage of the original screen area. To this end, the converted rectangle is enlarged so that when located in the appropriate NSView it completely covers the original rectangle (the left side of the figure below, with 15 degrees of rotation). In converting a size NSView simply treats it as a vector from (0.0, 0.0)

and maps it onto the destination coordinate system. Though the length remains the same, the balance along the two axes shifts according to the rotation (the right side of the figure below, rotated 45 degrees).



Drawing in an NSView

Drawing in an NSView is as simple as implementing the **drawRect:** method to generate the appropriate PostScript code for the image you want displayed—the display mechanism handles the rest of the work. On the other hand, it can be as complex as dealing with the PostScript language itself, the coordinate transformations from superview to subview, and the operation of the display mechanism. This section and "The Display Mechanism" progress from the basic to the esoteric, keeping the picture correct, if incomplete, at each stage.

In order for a concrete subclass of NSView to display any kind of image, it must implement the **drawRect:** method. This method is invoked during the display process to generate PostScript code that's rendered by the Window Server into a raster image. **drawRect:** takes a single argument, an NSRect describing the area that needs to be drawn in the receiver's own coordinate system. Here's an example:

```
- (void)drawRect:(NSRect)aRect
{
    PSsetgray(NSWhite);
    NSRectFill(aRect);

    PSsetgray(NSBlack);
    PSarc(0.0, 0.0, 117.0, 0.0, 360.0);
    PSfill();

    return;
}
```

This method first fills the view's background with white, then draws a black circle at the origin (0.0, 0.0). An NSView automatically clips drawing to its frame rectangle, so the results look like this:

Except for the background, this implementation of **drawRect:** ignores the rectangle provided, drawing everything each time it's invoked. This isn't a problem for a simple image, but for complex drawing it can be an extremely inefficient practice. Sending drawing instructions and data to the Window Server has a cost, and it's best to minimize that cost where possible. You can do this by testing whether a particular graphic shape intersects the rectangle being drawn, using **NSIntersectsRect()** and similar functions.

How to Draw

As indicated in the example above, drawing can be performed by invoking PostScript client library functions (also known as single-operator functions), which map directly to PostScript operators. The Application Kit provides a few higher-level mechanisms for handing PostScript instructions to the Window Server. The first is the **pswrap** program, which converts custom PostScript procedures into C functions that you can call in the same manner as client library functions. Wrapping complex drawing procedures minimizes the overhead of communication with the Window Server by passing a group of instructions in one interprocess message, as opposed to a number of such messages for repeated single-operator calls. The Application Kit itself defines some **pswrap** functions, such as **NSRectFill()**, and you can define your own.

Describing the PostScript language, client libraries, and **pswrap** is outside this scope of this class description. For more information, see:

> *PostScript Language Reference Manual, Second Edition*. Adobe Systems Incorporated. Addison Wesley, 1990. ISBN 0–201–18127–4.
>
> Descriptions of OPENSTEP PostScript operators and client functions, accessible from the Project Builder application in the Application Kit framework documentation.
>
> For information on **pswrap**, contact Adobe Systems.

The second higher-level mechanism is provided by Application Kit classes that perform drawing within an NSView, such as NSImage and the various NSCell subclasses. These classes send PostScript instructions to the Window Server but don't have the overhead of maintaining a drawing context that NSView has. Objects that draw themselves are useful for encapsulating graphic elements that need to be drawn over and over, at different locations, or in slightly different ways. See the appropriate class specifications for more information on drawing with them.

Another way of drawing within an NSView is to add subviews that each do their own drawing. This is somewhat more heavyweight than using NSCells or NSImages, but the elements of such a constructed group have the full power of the NSView machinery at their disposal, including the autosizing of components and event handling, features described later in this class description.

Checking the Output Device

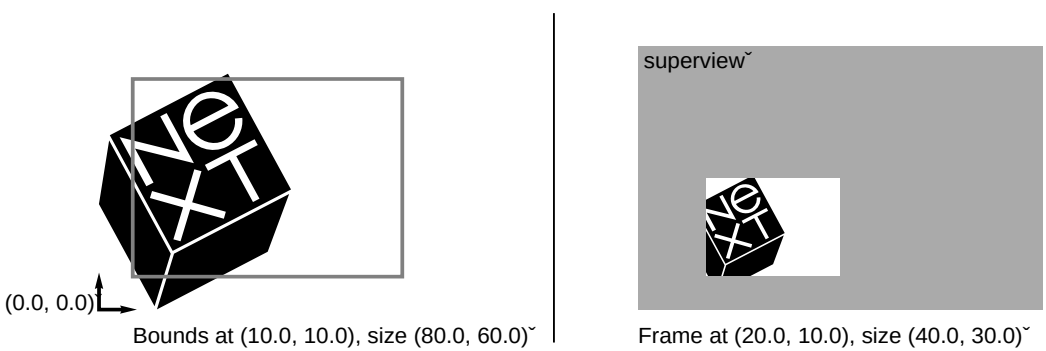
Most of an NSView's displayed image is a stable representation of its state, and is defined in the device-independent PostScript language. View objects also interact dynamically with the user, however, and this interaction often involves drawing that isn't integral to the image itself—selections and other highlighting, for example. Such drawing should be performed only to the computer screen, and never to a printer or fax device, or to the pasteboard (as when drawing an EPS image). You can predicate drawing on this difference of output device by sending the current DPS context an **isDrawingToScreen** message:

```
NSDPSContext *context = [NSDPSContext currentContext];

if (context && [context isDrawingToScreen]) {
    /* Draw things that should only appear on a computer screen. */
}
```

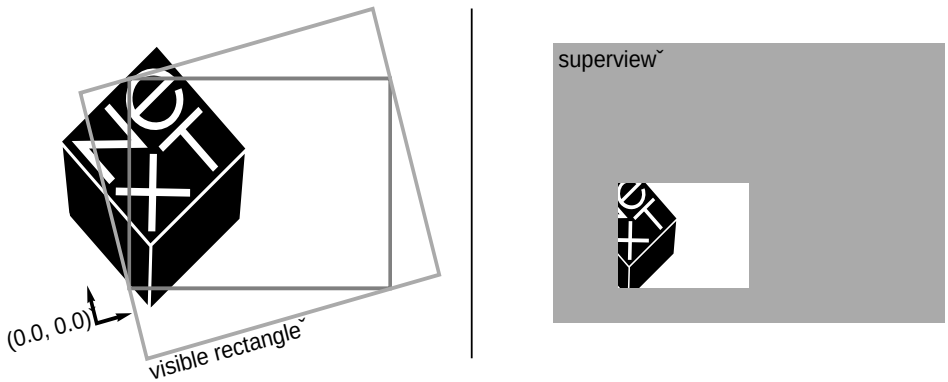
Coordinate System Transformations

By default, an NSView's coordinate system is based at (0.0, 0.0) in the lower-left corner of its bounds rectangle, its units are the same size as those of its superview, and its axes are parallel to those of its frame rectangle. To change this coordinate system you can alter the NSView's bounds rectangle, thereby placing the canvas inside the frame rectangle, or transform it directly using PostScript operators in the **drawRect:** method. Changing the bounds rectangle sets up the basic coordinate system, with which all drawing performed by the NSView begins; concrete subclasses of NSView typically alter the bounds rectangle immediately as needed in their **initWithFrame:** methods (or other designated initializers). Direct transformations are useful for temporary effects, such as scaling one axis to draw an oval instead of a circle, then scaling it back before stroking the path to preserve line widths; rotating the axes to draw text at an angle; or repeatedly translating the origin to draw the same figure in several locations.

The basic method for changing the bounds rectangle is **setBounds:**, which both positions and stretches the canvas. The origin of the rectangle provided to **setBounds:** becomes the lower-left corner of the bounds rectangle, and the size of the rectangle is made to fit in the frame rectangle, effectively scaling the NSView's drawn image. In the figure below, the bounds rectangle from the previous example is moved and doubled in size; the result appears on the right:



You can also set the parts of the bounds rectangle independently, using **setBoundsOrigin:** and **setBoundsSize:**. An additional method, **setBoundsRotation:**, rotates the coordinate system around its origin within the bounds rectangle (not the origin of the bounds rectangle itself). It also enlarges the visible rectangle to account for the rotation, so that it's expressed in the rotated coordinates yet completely covers the visible portion of the frame rectangle. This adds regions that must be drawn, yet will never be displayed (the triangular areas in the figure below). For this reason, rotating the bounds rectangle is strongly discouraged. It's better to rotate the coordinate system by using PostScript operators in the **drawRect:** method rather than by rotating the bounds rectangle.



setBoundsOrigin:, **setBoundsSize:**, and **setBoundsRotation:** all express their transformations in absolute terms. Another set of methods transform the coordinate system in relative terms; if you invoke them repeatedly, their effects accumulate. These methods are **translateOriginToPoint:**, **scaleUnitSquareToSize:**, and **rotateByAngle:**. See the individual method descriptions for more information.

One final type of coordinate transformation is statically established by overriding the **isFlipped** method. NSView's implementation returns NO, which means that the origin of the coordinate system lies at the lower-left corner of the default bounds rectangle and the *y* axis runs from bottom to top. When a subclass overrides this method to return YES, the NSView machinery automatically adjusts itself to assume that the upper-left corner of the NSView holds the origin. In other words, when **isFlipped** returns YES the *y* axis runs from top to bottom. A flipped coordinate system affects all drawing in the NSView itself and reckons the frame rectangles of all immediate subviews from their upper-left corners, but it doesn't affect the coordinate systems of those subviews or the drawing performed by them.

A flipped coordinate system doesn't affect an NSView's subviews, but the other coordinate transformations do. Translation of the bounds rectangle from the coordinate system origin shifts all subviews along with the rest of the NSView's image. Scaling and rotation actually affect the drawing of the subviews, as their coordinate systems inherit and build on these alterations. You can determine whether an NSView's coordinate system is (or was ever) altered from the base coordinate system of its window using two methods. **isRotatedFromBase** returns YES if the receiver or any of its ancestors in the view hierarchy has ever been rotated, whether of the frame or of the bounds rectangle. **isRotatedOrScaledFromBase** similarly returns YES if the receiver or any of its ancestors has ever been rotated or been scaled from the base

coordinate system's unit size. You can determine whether the NSView has never been rotated by checking that **isRotatedOrScaledFromBase** returns YES while **isRotatedFromBase** returns NO. Note that these methods only offer hints about the coordinate system. Their purpose is to help optimize certain operations, not to reflect the present state: Once an NSView is marked as having been rotated or scaled, it remains so marked for its lifetime.

To get the actual amount of rotation, use the **frameRotation** and **boundsRotation** methods. These return the rotation relative to the superview only, not to the base coordinate system, so if you want the latter amount you have to progress up through each superview to the NSWindow's content view, accumulating the rotation as you go. To get the scaling relative to the superview you can use **convertSize:toView:** and examine the ratio of the original size to that of the superview. To get the scaling relative to the base coordinate system, use **nil** as the second argument. This causes **convertSize:toView:** to convert to the NSWindow's base coordinate system.

The Display Mechanism

Displaying an NSView centers around the **drawRect:** method, which transmits drawing instructions to the Window Server. Before this can happen, however, a number of other things must be established. First, of course, is the rectangle in the view that needs to be drawn. Once this is known, the view must be checked for opacity; if the view is partially transparent, its nearest opaque ancestor must be found and drawing must commence from there. Once all of this is determined and a particular view is to be drawn, the Window Server must know which window device the view is in, how to clip drawing to the appropriate region, and what coordinate system to use. This is all handled outside **drawRect:**, by NSView's various display methods. The following sections examine each of these points in turn.

Marking a View as Needing Display

The most common way of causing an NSView to redisplay is to tell it that its image is invalid. On each pass through the event loop, all views that need to redisplay do so. NSView defines two methods for marking a view's image as invalid; **setNeedsDisplay:**, which invalidates the view's entire bounds rectangle, and **setNeedsDisplayInRect:**, which invalidates a portion of the view. The automatic display of views is controlled by their window; you can turn this behavior off using NSWindow's **setAutodisplay:** method. You should rarely need to do this however; the autodisplay mechanism is well-suited to most kinds of update and redisplay.

The autodisplay mechanism invokes various methods that actually do the work of displaying. You can also use these methods to force a view to redisplay itself immediately when necessary. **display** and **displayRect:** are the counterparts to the methods mentioned above; both cause the receiver to redisplay itself regardless of whether it needs to or not. Two additional methods, **displayIfNeeded** and **displayIfNeededInRect:**, redisplay invalidated rectangles in the receiver if it's been marked invalid with the methods above. The rectangles that actually get drawn are guaranteed to be at least those marked as invalid, but the view may coalesce them into larger rectangles to save multiple invocations of **drawRect:**.

Opacity

NSViews don't necessarily cover every bit of their frames with drawing. Because of this, the display methods must be sure to find an opaque background behind the view that's ostensibly being drawn, and begin displaying from there forward. The display methods above all pull back up the view hierarchy to the first view that responds YES to an **isOpaque** message, bringing the invalidated rectangles along. NSView by default responds NO to **isOpaque**, so it's important to remember to override this method to return YES if appropriate when defining a subclass. Most Application Kit subclasses of NSView actually do this.

If you want to exclude background views from drawing when forcing display to occur unconditionally, you can use NSView methods that explicitly omit backing up to an opaque ancestor. These methods, parallel to those mentioned above, are **displayRectIgnoringOpacity:**, **displayIfNeededIgnoringOpacity:**, and **displayIfNeededInRectIgnoringOpacity:**.

Locking Focus

Before a **display...** method invokes **drawRect:**, it sets the Window Server up with information about the view, including the window device it draws in, the coordinate system and clipping path it uses, and other PostScript graphics state (discussed in detail below, under "PostScript Graphics State Objects"). The method used to do this is **lockFocus**, and it has a companion method that undoes its effects, called **unlockFocus**.

All drawing code invoked by an NSView must be bracketed by invocations of these methods to produce proper results. If you define some methods that need to draw in a view without going through the display methods above, for example, you must send **lockFocus** to the view that you're drawing in before sending commands to the Window Server, and **unlockFocus** as soon as you are done.

It's perfectly reasonable to lock the PostScript focus on one view when another already has it. In fact, this is exactly what happens when subviews are drawn in their superview. The focusing machinery keeps a stack of which views have been focused, so that when one view is sent an **unlockFocus** message, the PostScript focus is restored to the view that was focused immediately before.

PostScript Graphics State Objects

When an NSView receives a **lockFocus** message, its basic drawing environment state is constructed and sent to the Window Server as a PostScript graphics state object, or *gstate* (this is a PostScript user object, not an Objective-C object). The basic state includes default values for parameters that don't change often, but leaves many other parameters undefined:

Parameter	Default Value
coordinate transformation	The NSView's coordinate system as established by the bounds rectangle
position	No default value, *must be set before drawing*

Parameter	Default Value
path	No default value
clipping path	As established by **lockFocus**
font	No default value, *must be set before drawing text*
line width	0.0
line cap	0 (a square butt end)
line join	0 (mitered joins)
halftone screen	A device-dependent, type 3 halftone dictionary
halftone phase	0,0
flatness	1.0
miter limit	10
dash pattern	A normal solid line
device	The current window
stroke adjust	true
color	No guaranteed default value
color space	No guaranteed default value, varies with color
color rendering	Calibrated RGB rendering
overprint	false
black generation	No default value
transfer	No default value
undercolor removal	No default value
alpha (opacity)	1.0 (opaque)
instance drawing mode	false

When drawing in an NSView, you must be sure to explicitly set relevant parameters that have no default value, or a PostScript error will result. Further, although drawing methods are free to set any gstate

parameter, they should always restore the parameters to their original values when finished. This protects multiple drawing methods, and objects that draw within an NSView, such as NSImages and NSCells, from altering each other's graphics states. You can protect the gstate by bracketing the changes with **PSgsave()** and **PSgrestore()**, or by explicitly placing the parameter in question on the stack and resetting it later—for example, saving the line width only using **PScurrentlinewidth()**, performing your drawing, then calling **PSsetlinewidth()** to restore the prior value.

Normally the graphics state object is reconstructed from scratch each time the NSView is focused. You can instruct an NSView to keep a graphics state object indefinitely by sending it an **allocateGState** message (typically in the initialization method for a concrete subclass). This eliminates the overhead of continual reconstruction of the graphics state, and also allows you to omit commands for setting parameters from your drawing code. However, because a graphics state object does consume a fair amount of memory, you should be sure to test your application's performance with and without it. Persistent gstate objects are most suitable for NSViews that must be redrawn frequently with the same parameters.

When you set an NSView to use a persistent gstate object, it doesn't actually allocate one until it needs it. When it does create the graphics state object, the NSView invokes its **setUpGState** method to set the parameters. Your subclass can override this method to establish the parameters that you want kept in the graphics state. Your version of **setUpGState** can use methods such as the **set** method found in NSColor and NSFont, as well as client library functions such as **PSsetlinewidth()**, **PSsetdash()**, and so on.

You can cause an NSView to discard its gstate object by sending it a **releaseGState** message, or cause the view simply to invalidate its gstate object by sending a **renewGState** message. The latter method causes the NSView to reestablish its gstate parameters by invoking **setUpGState** the next time it's needed. Finally, if for some reason you need to access the persistent gstate object directly, the **gState** method returns its PostScript user object identifier.Although applications rarely need to use this value, it can be passed to the few PostScript operators that take an object identifier as a parameter, such as **PScomposite** and **PSdissolve**.

Moving and Resizing NSViews

Repositioning an NSView is a potentially complex operation. Moving or resizing can expose portions of the NSView's superview that weren't previously visible, requiring the superview to redisplay. Resizing can also affect the layout of an NSView's subviews. Changes to an NSView's layout in any case may be of interest to other objects, which might need to be notified of the change. The following sections explore each of these areas.

Displaying After Moving or Resizing

None of the methods that alter an NSView's frame rectangle redisplays the NSView or marks it as needing display. When using the **setFrame...** methods, then, you must mark both the view being repositioned and its superview as needing display. This can be as simple as marking the superview in its entirety as needing display, or better, marking the superview in the old frame of the repositioned view and the view itself in its entirety. This code fragment sets **theView**'s frame rectangle, and updates its superview appropriately:

```
NSView *theView;      /* Assume this exists. */
NSRect newFrame;      /* Assume this exists. */

[[theView superview] setNeedsDisplayInRect:[theView frame]];
[theView setFrame:newFrame];
[theView setNeedsDisplay:YES];
```

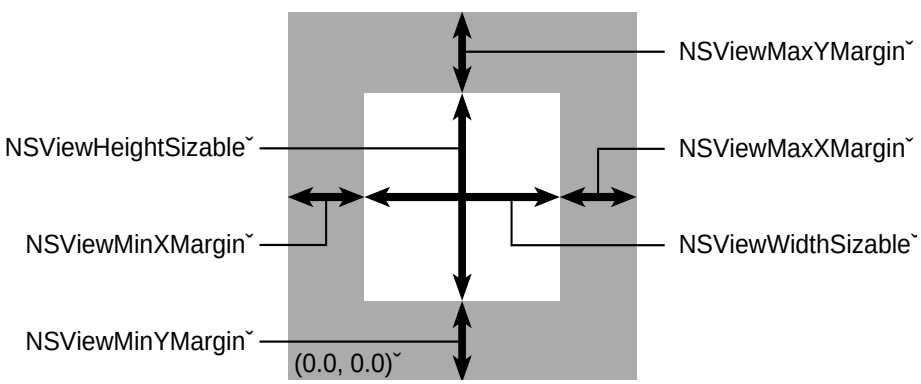
This sample marks the superview as needing display in the frame of the view about to be moved. Then, after **theView** is repositioned, it's marked as needing display in its entirety, which will nearly always be the case.

Note: The **setBounds...** methods also don't redisplay the NSView, but because their changes don't affect superviews you can simply mark the repositioned NSView as needing display.

The NSView class provides a mechanism to notify interested objects when a view object's frame rectangle is changed or its bounds rectangle is changed. The **setFrame...** methods post an NSViewFrameDidChangeNotification to the default notification center, unless the **setPostsFrameChangedNotification:** method has been used to turn notification off. The **setBounds...** methods post an NSViewBoundsDidChangeNotification to the default notification center, unless the **setPostsBoundsChangedNotification:** method has been used to turn notification off. Your application can use these notifications to perform special handling when a view object is moved or resized.

Autoresizing of Subviews

When an NSView's frame size changes, the layout of its subviews must often be adjusted to fit in the new size. NSView defines a mechanism that automates this process, allowing you to specify how any NSView should reposition itself when its superview is resized. Interface Builder allows you to set these attributes graphically with its Size Inspector, and in test mode you can examine the effects of autoresizing. You can also set autoresizing attributes programmatically using **setAutoresizingMask:** with a mask containing any of the constants illustrated below, combined using the C bitwise OR operator:



When one of these mask flags is omitted, the NSView's layout is fixed in that aspect; when it's included the NSView's layout is flexible in that aspect. For example, to keep an NSView in the lower left corner of its

superview, you specify NSViewMaxXMargin | NSViewMaxYMargin. When more than one aspect along an axis is made flexible, the resize amount is distributed evenly among them.

Autoresizing is on by default, but you can turn it off using the **setAutoresizesSubviews:** method. Note that when you turn off an NSView's autoresizing, all of its descendants are likewise shielded from changes in the superview. Changes to subviews, however, can still percolate downward. Similarly, if a subview has no autoresize mask, it won't change in size, and therefore none of its subview will autoresize.

Autoresizing is accomplished using two methods. **resizeSubviewsWithOldSize:** is invoked automatically by an NSView whenever its frame size changes. This method then simply sends a **resizeWithOldSuperviewSize:** message to each subview. Each subview compares the old frame size to the new size and adjusts its position and size according to its autoresize mask. Subclasses of NSView can override either method to alter their autoresizing behavior.

Two cautions apply to autoresizing. First, it doesn't work at all in NSViews that have been rotated. Subviews that have been rotated can autoresize within a nonaltered superview, but then their descendants aren't autoresized. Also, for autoresizing to work correctly, the subview being autoresized must lie completely within its superview's frame. Apart from these limitations, autoresizing covers most layout changes quite well.

Notifications

Beyond resizing its subviews, an NSView broadcasts notifications to interested observers any time its bounds and frame rectangles change. The notification names are NSViewFrameDidChangeNotification and NSBoundsDidChangeNotification, respectively. An NSView that bases its own display on the layout of its subviews, for example, can register itself as an observer for those subviews and update itself any time they're moved or resized. NSScrollView and NSClipView cooperate in this manner to adjust the NSScrollView's NSScrollers. You can turn notifications on and off using **setPostsFrameChangedNotification:** and **setPostsBoundsChangedNofitications:**.

Event Handling

NSViews are the most typical receivers of event and action messages, as described in the NSResponder and NSEvent class specifications. An NSView subclass can handle any event or action message simply by implementing it (being sure to invoke **super**'s implementation as needed). Then, if an instance of that class is the first in the responder chain to respond to that message, it receives such messages as they're generated.

Except for an NSWindow's content view, an NSView's next responder is always its superview—most of the responder chain, in fact, comprises the NSViews from an NSWindow's first responder up to its content view. NSView **addSubview:** method automatically sets the receiver as the new subview's superview; you should never send **setNextResponder:** to an NSView object. You can safely add responders to the top end of an NSWindow's responder chain—the NSWindow itself if it has no delegate, or the delegate if it does.

As the class that handles display, NSView is the typical recipient of mouse and keyboard events. Mouse clicks, drags, and movements usually occur in some NSView or other, and most keystrokes represent text to be added for display at some point in a window. A mouse event starts at the lowest NSView containing

it in the view hierarchy (or, the topmost NSView displayed under the cursor), and proceeds up the responder chain through superviews until some object handles it. “Mouse Events” below covers the details of handling mouse events. Most keyboard events start at the first responder, whatever it might be, and are similarly offered up the responder chain. Some actually change the first responder, thus allowing the user to perform many actions without using the mouse. See the NSResponder class specification for information on keyboard events. Tracking-rectangle events are monitored by the NSWindow and dispatched directly to the object that owns the tracking rectangle. “Tracking Rectangles and Cursor Rectangles” describes how to set up and handle these. An additional section covers the use of context-sensitive pop-up menus by your views.

Mouse Events

An NSView can receive mouse events of three general types: clicks, drags, and movements. A custom subclass of NSView can interpret a mouse event as a cue to perform a certain action, such as sending a target-action message, selecting a graphic element, and so on. NSViews automatically receive mouse-clicked and mouse-dragged events, but because mouse-moved events occur so often and can bog down the event queue, an NSView must explicitly request its NSWindow to watch for them using NSWindow’s **setAcceptsMouseMovedEvents:** method. Tracking rectangles, described below, are a less expensive way of following the mouse’s location.

The NSView selected to receive a mouse event is determined by the NSWindow using NSView’s **hitTest:** method, which returns the lowest descendant that contains the cursor location of the event (this is also the topmost NSView displayed). Once the recipient is determined, the NSWindow sends it a **mouseDown:** message, which includes an NSEvent object containing information about the click. NSEvent’s **locationInWindow** locates the cursor’s hot spot in the coordinate system of the receiver’s NSWindow. To convert it to the NSView’s coordinate system, use **convertPoint:fromView:** with a **nil** NSView argument. From here, you can use **mouse:inRect:** to determine whether the click occurred in an interesting area.

One of the earliest things to consider in handling mouse-down events is whether the receiving NSView should become the first responder, which means that it will be the first candidate for subsequent key events and action messages. NSViews that handle graphic elements that the user can select—drawing shapes or text, for example—should typically accept first responder status on a mouse-down event, by overriding the **acceptsFirstResponder** method to return YES. This results in the window making the receiving NSView first responder with NSWindow’s **makeFirstResponder:** method. Some NSViews, however, may not wish to change the selection upon the first mouse click in a non-key window, which should normally only order the window to the front. NSView’s **acceptsFirstMouse:** method controls whether an initial mouse click is sent to the NSView or not. By default it returns NO, which in most cases is appropriate behavior. Certain subclasses, such as controls that don’t affect the selection, override this method to return YES.

Once an NSView has accepted a mouse event and determined its location, it can also check which mouse button was clicked and how many times. NSEvent’s **type** method distinguishes between left and right mouse events, and the NSView can base its behavior on this information. Right mouse events are defined by the Application Kit to open pop-up menus, but you can override this behavior if necessary. NSEvent’s **clickCount** method returns a number identifying the mouse event as a single-, double-, or triple-click (and so on).

NSViews that handle mouse clicks as a single event, from mouse down, through dragging, to mouse up, must usually short-circuit the application's normal event loop, entering a *modal event loop* to catch and process only events of interest. For example, an NSButton highlights upon a mouse-down event, then follows the mouse location during dragging, highlighting when the mouse is inside and unhighlighting when the mouse is outside. If the mouse is inside on the mouse-up event, the NSButton sends its action message. This method template shows one possible kind of modal event loop:

```
- (void)mouseDown:(NSEvent *)theEvent
{
    BOOL keepOn = YES;
    BOOL isInside = YES;
    NSPoint mouseLoc;

    do {
         mouseLoc = [self convertPoint:[theEvent mouseLocationInWindow
            fromView:nil]];
         isInside = [self mouse:mouseLoc inRect:[self bounds]];

         switch ([theEvent type]) {
            case NSLeftMouseDragged:
                   [self highlight:isInside];
                   break;
            case NSLeftMouseUp:
                   if (isInside) [self doSomethingSignificant];
                   [self highlight:NO];
                   keepOn = NO;
                   break;
            default:
                   /* Ignore any other kind of event. */
                   break;
        }

        theEvent = [[self window] nextEventMatchingMask: NSLeftMouseUpMask |
               NSLeftMouseDraggedMask];

     }while (keepOn);

    return;
}
```

This loop converts the mouse location and checks whether it's inside the receiver. It highlights itself using the fictional **highlight:** method according to this, and on a mouse up inside, invokes **doSomethingSignificant** to perform an important action. Instead of merely highlighting, a custom NSView might move a selected object, draw a graphic image according to the mouse's location, and so on.

This kind of modal event loop is driven only as long as the user actually moves the mouse. It won't work, for example, to cause continual scrolling if the user presses the mouse button but never moves the mouse

itself. For this, your modal loop should start a periodic event stream using NSEvent's class method **startPeriodicEventsAfterDelay:withPeriod:**, and add NSPeriodicMask to the mask passed to **nextEventMatchingMask:**. In the **switch()** statement the NSView can then check for a case of NSPeriodic and take whatever action it needs to; scrolling a document view or moving a step in an animation, for example. If you need to check the mouse location during a periodic event, you can use NSWindow's **mouseLocationOutsideOfEventStream** method.

Tracking Rectangles and Cursor Rectangles

One special type of event is that for tracking mouse movement into and out of a region in the NSView. Such a region is known as a *tracking rectangle*; it triggers mouse-entered events when the cursor enters it and mouse-exited events when the cursor leaves it. This can be useful for displaying context-sensitive messages or highlighting graphic elements under the cursor, for example. An NSView can have any number of tracking rectangles, which can overlap or be nested one within the other; the NSEvent objects generated for tracking events include a tag that identifies the rectangle that triggered the event.

To create a tracking rectangle, use the **addTrackingRect:owner:userData:assumeInside:** method. This method registers an owner for the tracking rectangle provided, so that the owner receives the event messages. This is typically the NSView itself, but need not be. The method returns the tracking rectangle's tag so that you can store it for later reference in the event handling methods, **mouseEntered:** and **mouseExited:**. To remove a tracking rectangle, use the **removeTrackingRect:** method, which takes as an argument the tag of the tracking rectangle to remove.

Tracking rectangles, though created and used by NSViews, are actually maintained by NSWindows. Because of this, a tracking rectangle is a static entity; it doesn't move or change its size when the NSView does. If you use tracking rectangles, you should be sure to remove and reestablish them any time you change the frame rectangle of the NSView that contains them. If you're using a custom subclass of NSView, you can override the frame- and bounds-setting methods to do this. You can also register an observer for the NSViewFrameDidChangeNotification (described below), and have it reestablish the tracking rectangles on receiving the notification.

One common use of tracking rectangles is to change the cursor image over different types of graphic elements. Text, for example, typically requires an I-beam cursor. Changing the cursor is such a common operation that NSView defines several convenience methods to ease the process. A tracking rectangle generated by these methods is called a *cursor rectangle*. The Application Kit itself assumes ownership of cursor rectangles, so that when the user moves the mouse over the rectangle the cursor automatically changes to the appropriate image. Unlike general tracking rectangles, cursor rectangles may not partially overlap. They may, however, be completely nested, one within the other.

Because cursor rectangles need to be reset often as the NSView's size and graphic elements change, NSView defines a single method, **resetCursorRects**, that's invoked any time its cursor rectangles need to be reestablished. A concrete subclass overrides this method, invoking **addCursorRect:cursor:** for each cursor rectangle it wishes to set. Thereafter, the NSView's cursor rectangles can be rebuilt by invoking NSWindow's **invalidateCursorRectsForView:** method. If you find you need to temporarily remove a

single cursor rectangle, you can do this with **removeCursorRect:cursor:**. Be aware that **resetCursorRects** will reestablish that rectangle, unless you implement it to do otherwise.

An NSView's cursor rectangles are automatically reset whenever:

- Its frame or bounds rectangle changes, whether by a **setFrame...** or **setBounds...** message or by autoresizing.
- Its NSWindow is resized. In this case all of the NSWindow's view objects get their cursor rectangles reset.
- It's moved in the view hierarchy.
- It's scrolled in an NSScrollView or NSClipView.

You can temporarily disable all the cursor rectangles in a window using NSWindow's **disableCursorRects** method or enable them with the **enableCursorRects** method. NSWindow's **areCursorRectsEnabled** method tells you whether they're currently enabled.

Context-Sensitive Menus

On Microsoft Windows, any view can be assigned a pop-up menu that's displayed when the user clicks the right mouse button over the view. **setMenu:** assigns an NSMenu to a view, and **menu** returns it. Your subclass can define a menu that's used for all instances by implementing the **defaultMenu** class method. It can also change the menu displayed based on the mouse event by overriding the **menuForEvent:** instance method. This allows the view clicked to display different menus based on the location of the mouse and of the view's state, or to change or enable individual menu items based on the commands available for the view or for that region of the view. See the NSMenu and NSMenuItem class and protocol specifications for more information on using menus.

Tooltips

A tooltip is a bit of text that provides information about a view. If the user holds the cursor over the view for more than the default delay, the tooltip text is displayed in a small framed rectangle next to the cursor. By default, a view does not display a tooltip. To turn on tooltip display for a view, you invoke the **setToolTip:** method to install tooltip text for the view. To turn display off, you invoke **setToolTip:** with a **nil** string.

Printing and Faxing

Printing or faxing an NSView uses the same PostScript description as for displaying on the screen, by simply changing the device. An NSView can check whether it's drawing to the screen in order to conditionally include or omit elements such as highlighting, but normally doesn't need to be involved with the PostScript generation process in a special way for printing. It may, however, need to take part in peripheral issues, including how it's divided into pages and placed on them, and generation of document structuring comments used by some PostScript document programs. The sections below cover these areas.

To print or fax an NSView, send it a **print:** or **fax:** message. You can also generate an EPS representation using either **dataWithEPSInsideRect:** or **writeEPSInsideRect:toPasteboard:**. For any of these jobs, the NSView creates an NSPrintOperation object that manages the process of generating proper PostScript code for a printer or fax device. NSPageLayout, NSPrintInfo, and NSPrintPanel objects are also involved in the process. See those classes' specifications for more information on the printing process itself.

Pagination

When an NSView is printed onto pages smaller than itself, it tiles itself out onto separate logical pages so that its entire visible region is printed. A subclass of NSView can alter the way pagination is performed by overriding two small sets of methods. The first set affects automatic pagination; the second replaces automatic pagination completely. One extra method allows the NSView to adjust the location of the printed image on the page. Finally, after pagination has actually been performed, the NSView is given the chance to draw additional marks on the page.

NSView's automatic pagination tries to fit as much of the view being printed onto a logical page, slicing the view into the largest possible chunks. This is sufficient for many views, but if a view's image must be divided only at certain places—between lines of text or cells in a table, for example, the view can adjust the automatic mechanism to accommodate this by reducing the height or width of each page. It does so by overriding up to four methods. **adjustPageHeightNew:top:bottom:limit:** provides an out parameter for the new bottom coordinate of the page, followed by the proposed top and bottom. An additional parameter limits the height of the page; the bottom can't be moved above it. **adjustPageWidthNew:left:right:limit:** works in the same way to allow the view to adjust the width of a page. The limits are calculated as a percentage of the proposed page's height or width. Your view subclass can also customize this percentage by overriding the methods **heightAdjustLimit** and **widthAdjustLimit** to return the reducible fraction of the page.

More complex views, such as those that display separate pages over a background, need to direct their own pagination. An NSView subclass that needs to do so overrides the **knowsPagesFirst:last:** method to return YES, which signals that it will be calculating each page's dimensions, and returns by reference its first and last page numbers. The pagination machinery then uses these numbers, sending **rectForPage:** to the NSView, which uses the page number and the current printing information to calculate an appropriate rectangle in its coordinate system. The **adjustPage...** methods aren't used in this case.

The last stage of pagination involves placing the image to be printed on the logical page. NSView's **locationOfPrintRect:** places it according to the NSPrintInfo's status. By default it places the image in the upper left corner of the page, but if NSPrintInfo's **isHorizontallyCentered** or **isVerticallyCentered** methods return YES, it centers a single-page image along the appropriate axis. A multiple-page document, however, is always placed so that the divided pieces can be assembled at their edges.

After the NSView has sliced out a rectangle and positioned it on a page, it's given two chances to add extra marks to the page, such as crop marks or fold lines. **drawPageBorderWithSize:** is used for logical pages, and is invoked for each paginated portion of the view. **drawSheetBorderWithSize: is used for actual physical pages, or sheets, on which one or more logical pages may be laid out.** In a 2-up printing, for example, the former method is invoked twice for each sheet, while the latter is invoked once for each sheet.

PostScript Document Structure

As an adjunct to the PostScript language itself, Adobe has defined a set of *document structuring conventions* that describe the internal structure of a given PostScript language document. NSView properly generates the basic information needed to structure its output, and defines a number of methods that subclasses can override to provide additional information. This section only describes the methods that relate to the structure of a conforming PostScript language document; see the individual method descriptions and Adobe's *PostScript Language Reference Manual*, Appendix G for more information.

An NSView subclass can override any of the methods that write out document structuring comments and definitions. When overriding **begin...** or **add...** methods, be sure to invoke **super**'s implementation *before* writing additional information; when overriding **end...** methods, invoke **super**'s implementation last. This sample method, for example, adds a comment to the header of a document:

```
- (void)endHeaderComments
{
    NSDPSContext *context = [NSDPSContext currentContext];
    [context printFormat:@"%%%%SomeComment: %d\n", someNumber];

    [super endHeaderComments];
    return;
}
```

The initial portion of a conforming PostScript language document is called the *prologue*, and contains two parts itself: the header and a set of procedure definitions. NSView's **beginPrologueBBox:...** writes out the very beginning of the document. **endHeaderComments** closes the first part of the prologue. Subclasses can add their own procedure definitions to the end of the prologue by overriding **endPrologue**.

After the prologue comes the *script*, which contains a section that applies to the entire document, followed by sections for each page, and finally the document trailer. **beginSetup** and **endSetup** write the document setup section. Each page is written with five methods, in addition to **drawRect:**. **beginPage:label:bBox: fonts:** writes out the beginning of each page's document structuring comments. It's followed by **beginPageSetupRect:placement:**, which starts the page setup section. An additional method, **addToPageSetup**, does nothing by default, but allows subclasses to append extra procedure definitions and comments to the page setup. The page setup concludes with an **endPageSetup** message. After all this, **endPage** wraps up the page description; subclasses can override this method to add document structuring comments and PostScript code to the page trailer. The document trailer is written by the **beginTrailer** and **endTrailer** methods.

Communicating with the Window Server During Printing

While an NSView is printing, its connection to the Window Server is replaced by a connection to the print job output. Sometimes the NSView needs to communicate briefly with the Window Server while printing; for example, it may need to read some data stored only on the Window Server, or open an attention panel to alert the user of a problem. In these cases, it can temporarily swap in the NSApplication object's DPS

context to restore access to the application's Window Server state and to the screen. When finished, the view object restores the print operation's context to continue generating its image:

```
[NSDPSContext setCurrentContext:[NSApp context]];
/* Communicate with the Window Server. */

[NSDPSContext setCurrentContext:[[NSPrintOperation currentOperation] context]];
/* Resume generating PostScript code. */
```

Other Features

Besides the fundamentals of drawing and event handling, NSView includes several auxiliary features. These are tagging NSViews for quick location, support for dragging of images and file icons, and cooperation with the scrolling machinery to facilitate viewing larger NSViews through smaller ones. The following sections introduce each of these features and name the methods and cooperating classes or protocols involved in each.

Tags

NSView defines methods that allow you to tag individual view objects with integer tags and to search the view hierarchy based on those tags. NSView's **tag** method always returns –1. You can override this in subclasses to return a special value, or even add a **setTag:** method to allow the tag to be changed at run time (several Application Kit classes, especially NSControl and NSCell, do just this). The **viewWithTag:** method proceeds through all of the receiver's descendants (including itself), searching for a subview with the given tag and returning it if it's found.

Dragging

A view object can act as either the source or destination for dragged images and file icons. The basic dragging methods, **dragImage:...** and **dragFile:...** methods, handle the mechanics of moving the image on the screen and notifying the destination of the dragging operations. To act as a source for dragging operations, a concrete subclass of NSView can adopt the NSDraggingSource protocol, by which the source indicates what kinds of dragging operations are allowed and is notified of dragging operations as they begin. Both NSView and NSWindow subclasses can act as destinations for dragging operations, by adopting the NSDraggingDestination protocol and making use of the NSDraggingInfo protocol. For more information see the dragging protocol specifications and the descriptions of **dragImage:...** and **dragFile:...** in this specification.

Scrolling

NSView defines a number of methods to support scrolling, whereby the NSView being scrolled—the *document view*—is displayed partially through another—the *content* or *clip view* (not to be confused with a window's content view). Scrolling is effected by moving the clip view's bounds rectangle, which reveals the different regions of the document view. Most of the scrolling methods assume that the NSView is

enclosed within an NSClipView and an NSScrollView, which handle the mechanics of scrolling for you. You can, however, reproduce the effects of scrolling yourself if you wish. See the NSScrollView, NSClipView, and NSScroller class specifications for information on how scrolling is implemented by the Application Kit.

NSView's most direct scrolling methods are **scrollPoint:** and **scrollRectToVisible:**, both of which assume that the receiver is embedded in an NSClipView. These methods move the clip view so that the requested point or rectangle in the receiver become visible. Another method, **autoscroll:**, automatically scrolls the receiver in an NSClipView based on the location of the mouse. It's useful for moving the document view when the user drags an icon outside of the visible area. The **enclosingScrollView** method returns the NSScrollView that contains the NSView, allowing you to tune the way scrolling occurs.

Two other methods aid in scrolling. A subclass of NSView can override **adjustScroll:** to change the way automatic (user-driven) scrolling occurs. It can quantize scrolling into regular units, to the edges of a spreadsheet's cells, for example, or simply limit scrolling to a specific region of the NSView. The last scrolling method, **scrollRect:by:**, copies an already-drawn portion of the NSView to a new location. It's useful for producing temporary effects, but note that any subsequent drawing will obliterate the copied portion.

Method Types

Creating instances

– initWithFrame:

Managing the view hierarchy

– superview
– subviews
– window
– addSubview:
– addSubview:positioned:relativeTo:
– removeFromSuperview
– removeFromSuperviewWithoutNeedingDisplay
– replaceSubview:with:
– isDescendantOf:
– opaqueAncestor
– ancestorSharedWithView:
– sortSubviewsUsingFunction:context:
– viewWillMoveToSuperview:
– viewWillMoveToWindow:

Searching by tag

– viewWithTag:
– tag

Modifying the frame rectangle

– setFrame:
– frame
– setFrameOrigin:
– setFrameSize:
– setFrameRotation:
– frameRotation

Modifying the bounds rectangle

– setBounds:
– bounds
– setBoundsOrigin:
– setBoundsSize:
– setBoundsRotation:
– boundsRotation

Modifying the coordinate system

– translateOriginToPoint:
– scaleUnitSquareToSize:
– rotateByAngle:

Examining coordinate system modifications

– isFlipped
– isRotatedFromBase
– isRotatedOrScaledFromBase

Converting coordinates

– convertPoint:fromView:
– convertPoint:toView:
– convertSize:fromView:
– convertSize:toView:
– convertRect:fromView:
– convertRect:toView:
– centerScanRect:

Controlling notifications

– setPostsFrameChangedNotifications:
– postsFrameChangedNotifications
– setPostsBoundsChangedNotifications:
– postsBoundsChangedNotifications

Resizing subviews

– resizeSubviewsWithOldSize:
– resizeWithOldSuperviewSize:
– setAutoresizesSubviews:
– autoresizesSubviews
– setAutoresizingMask:
– autoresizingMask

Focusing

– lockFocus
– unlockFocus
+ focusView

Displaying

– setNeedsDisplay:
– setNeedsDisplayInRect:
– needsDisplay
– display
– displayRect:
– displayRectIgnoringOpacity:
– displayIfNeeded
– displayIfNeededInRect:
– displayIfNeededIgnoringOpacity
– displayIfNeededInRectIgnoringOpacity:
– isOpaque

Drawing

– drawRect:
– visibleRect
– canDraw
– shouldDrawColor

Setting the interface style

– setInterfaceStyle:
– interfaceStyle

Managing a graphics state

– allocateGState
– gState
– setUpGState
– renewGState
– releaseGState

Event handling

– acceptsFirstMouse:
– hitTest:
– mouse:inRect:
– performKeyEquivalent:
– performMnemonic:

Dragging operations

– dragImage:at:offset:event:pasteboard:source:slideBack:
– dragFile:fromRect:slideBack:event:
– registerForDraggedTypes:
– unregisterDraggedTypes
– shouldDelayWindowOrderingForEvent:

Managing cursor rectangles

– addCursorRect:cursor:
– removeCursorRect:cursor:
– discardCursorRects
– resetCursorRects

Managing tool tips

– setToolTip:
– toolTip

Managing tracking rectangles

– addTrackingRect:owner:userData:assumeInside:
– removeTrackingRect:

Scrolling

– scrollPoint:
– scrollRectToVisible:
– autoscroll:
– adjustScroll:
– scrollRect:by:
– enclosingScrollView
– scrollClipView:toPoint:
– reflectScrolledClipView:

Context-sensitive menus

– menuForEvent:
+ defaultMenu

Managing the key view loop

– setNextKeyView:
– nextKeyView
– nextValidKeyView
– previousKeyView
– previousValidKeyView

Printing and faxing

– print:
– fax:
– dataWithEPSInsideRect:
– writeEPSInsideRect:toPasteboard:

Pagination

– heightAdjustLimit
– widthAdjustLimit
– adjustPageWidthNew:left:right:limit:
– adjustPageHeightNew:top:bottom:limit:
– knowsPagesFirst:last:
– rectForPage:
– locationOfPrintRect:

Adorning pages in printout

– drawPageBorderWithSize:
– drawSheetBorderWithSize:

Writing conforming PostScript

– beginPrologueBBox:creationDate:createdBy:fonts:
forWhom:pages:title:
– endHeaderComments
– endPrologue
– beginSetup
– endSetup
– beginPage:label:bBox:fonts:
– beginPageSetupRect:placement:
– addToPageSetup
– endPageSetup
– endPage
– beginTrailer
– endTrailer

Class Methods

defaultMenu

+ (NSMenu *)**defaultMenu**

Overridden by subclasses to return the default pop-up menu for instances of the receiving class. NSView's implementation returns **nil**. This menu is used only on Microsoft Windows.

See also: – **menuForEvent:**, – **menu**(NSResponder)

focusView

+ (NSView *)**focusView**

Returns the currently focused NSView object, or **nil** if there is none.

See also: – **lockFocus**, – **unlockFocus**

Instance Methods

acceptsFirstMouse:

– (BOOL)**acceptsFirstMouse:**(NSEvent *)*theEvent*

Overridden by subclasses to return YES if the receiver should be sent a **mouseDown:** message for *theEvent*, an initial mouse-down event over the receiver in its window, NO if not. The receiver can either return a value unconditionally, or use *theEvent*'s location to determine whether or not it wants the event. NSView's implementation ignores *theEvent* and returns NO.

Override this method in a subclass to allow instances to respond to initial mouse-down events. For example, most view objects refuse an initial mouse-down event, so that the event simply activates the window. Many control objects, however, such as NSButton and NSSlider, do accept them, so that the user can immediately manipulate the control without having to release the mouse button.

See also: – **hitTest:**

addCursorRect:cursor:

– (void)**addCursorRect:**(NSRect)*aRect* **cursor:**(NSCursor *)*aCursor*

Establishes *aCursor* as the cursor to be used when the mouse pointer lies within *aRect*.

Note: Cursor rectangles aren't subject to clipping by superviews, nor are they intended for use with rotated NSViews. You should explicitly confine a cursor rectangle to the NSView's visible rectangle to prevent improper behavior.

This method is intended to be invoked only by the **resetCursorRects** method. If invoked in any other way, the resulting cursor rectangle will be discarded the next time the NSView's cursor rectangles are rebuilt.

See also: – **removeCursorRect:cursor:**, – **discardCursorRects**, – **resetCursorRects**, – **visibleRect**

addSubview:

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

Adds *aView* to the receiver's subviews so that it's displayed above its siblings. Also sets the receiver as *aView*'s next responder.

The receiver retains *aView*. If you use **removeFromSuperview** to remove *aView* from the view hierarchy, *aView* is released. If you want to keep using *aView* after removing it from the view hierarchy (if, for example, you are swapping through a number of views), you must retain it before invoking **removeFromSuperview**.

See also: – **addSubview:positioned:relativeTo:**, – **subviews**, – **removeFromSuperview**, – **setNextResponder:** (NSResponder), – **viewWillMoveToSuperview:**, – **viewWillMoveToWindow:**

addSubview:positioned:relativeTo:

– (void)**addSubview:**(NSView *)*aView*
positioned:(NSWindowOrderingMode)*place*
relativeTo:(NSView *)*otherView*

Inserts *aView* among the receiver's subviews so that it's displayed immediately above or below *otherView* according to whether *place* is NSWindowAbove or NSWindowBelow. If *otherView* is **nil** (or isn't a subview of the receiver), *aView* is added above or below all of its new siblings. Also sets the receiver as *aView*'s next responder.

The receiver retains *aView*. If you use **removeFromSuperview** to remove *aView* from the view hierarchy, *aView* is released. If you want to keep using *aView* after removing it from the view hierarchy (if, for example, you are swapping through a number of views), you must retain it before invoking **removeFromSuperview**.

See also: – **addSubview:**, – **subviews**, – **removeFromSuperview**, – **setNextResponder:** (NSResponder)

addToPageSetup

– (void)**addToPageSetup**

Implemented by subclasses that perform their own pagination to add a scaling operator to the PostScript code generated when printing. This method is invoked by **print:** and **fax:**. NSView's implementation of this method does nothing.

See the NSPrintInfo class specification for information on retrieving document scaling during printing.

See also: – **beginPageSetupRect:placement:**

addTrackingRect:owner:userData:assumeInside:

– (NSTrackingRectTag)**addTrackingRect:**(NSRect)*aRect*
owner:(id)*anObject*
userData:(void *)*userData*
assumeInside:(BOOL)*flag*

Establishes *aRect* as an area for tracking mouse-entered and mouse-exited events within the receiver, and returns an tag that identifies the tracking rectangle in NSEvent objects and that can be used to remove the tracking rectangle. *anObject* is the object that gets sent the event messages. It can be the receiver itself or some other object (such as an NSCursor or a custom drawing tool object), as long as it responds to both **mouseEntered:** and **mouseExited:**. *userData* is supplied in the NSEvent object for each tracking event. *flag* determines which event is sent first by indicating where the mouse is assumed to be at the time this method is invoked. If *flag* is YES, the first event will be generated when the mouse leaves *aRect*; if *flag* is NO the first event will be generated when the mouse enters it.

Tracking rectangles provide a general mechanism that can be used to trigger actions based on the mouse location (for example, a status bar or hint field that provides information on the item the cursor lies over). To simply change the cursor over a particular area, use **addCursorRect:cursor:**. If you must use tracking rectangles to change the cursor, the NSCursor class specification describes the additional methods that must be invoked to change cursors by using tracking rectangles.

See also: – **removeTrackingRect:**, – **userData** (NSEvent)

adjustPageHeightNew:top:bottom:limit:

– (void)**adjustPageHeightNew:**(float *)*newBottom*
top:(float)*top*
bottom:(float)*proposedBottom*
limit:(float)*bottomLimit*

Overridden by subclasses to adjust page height during automatic pagination. This method is invoked by **print:** and **fax:** with *top* and *proposedBottom* set to the top and bottom edges of the pending page rectangle in the receiver's coordinate system. The receiver can raise the bottom edge and return the new value in *newBottom*, allowing it to prevent items such as lines of text from being divided across pages. *bottomLimit* is the topmost value that *newBottom* can be set to, as calculated using the return value of **heightAdjustLimit**. If this limit is exceeded, the pagination mechanism simply uses *bottomLimit* for the bottom edge.

NSView's implementation of this method propagates the message to its subviews, allowing nested views to adjust page height for their drawing as well. An NSButton or other small view, for example, will nudge the bottom edge up if necessary to prevent itself from being cut in two (thereby pushing it onto an adjacent page). Subclasses should invoke **super**'s implementation, if desired, after first making their own adjustments.

See also: – **adjustPageWidthNew:left:right:limit:**

adjustPageWidthNew:left:right:limit:

– (void)**adjustPageWidthNew:**(float *)*newRight*
left:(float)*left*
right:(float)*proposedRight*
limit:(float)*rightLimit*

Overridden by subclasses to adjust page width during automatic pagination. This method is invoked by **print:** and **fax:** with *left* and *proposedRight* set to the side edges of the pending page rectangle in the receiver's coordinate system. The receiver can pull in the right edge and return the new value in *newRight*, allowing it to prevent items such as small images or text columns from being divided across pages. *rightLimit* is the leftmost value that *newRight* can be set to, as calculated using the return value of **widthAdjustLimit**. If this limit is exceeded, the pagination mechanism simply uses *rightLimit* for the right edge.

NSView's implementation of this method propagates the message to its subviews, allowing nested views to adjust page width for their drawing as well. An NSButton or other small view, for example, will nudge the bottom edge up if necessary to prevent itself from being cut in two (thereby pushing it onto an adjacent page). Subclasses should invoke **super**'s implementation, if desired, after first making their own adjustments.

See also: – **adjustPageHeightNew:top:bottom:limit:**

adjustScroll:

– (NSRect)**adjustScroll:**(NSRect)*proposedVisibleRect*

Overridden by subclasses to modify *proposedVisibleRect*, returning the altered rectangle. NSClipView invokes this method to allow its document view to adjust its position during scrolling. For example, a custom view object that displays a table of data can adjust the origin of *proposedVisibleRect* so that rows or columns aren't cut off by the edge of the enclosing NSClipView. NSView's implementation simply returns *proposedVisibleRect*.

Note: NSClipView only invokes this method during automatic or user-controlled scrolling. Its **scrollToPoint:** method doesn't invoke this method, so you can still force a scroll to an arbitrary point.

allocateGState

– (void)**allocateGState**

Causes the receiver to maintain a private PostScript graphics state object, which encapsulates all parameters of the graphics environment. If you do not invoke **allocateGState**, a graphics state object is constructed from scratch each time the NSView is focused.

The receiver builds the graphics state parameters using **setUpGState**, then automatically establishes this graphics state each time the PostScript focus is locked on it. A graphics state may improve performance for

view objects that are focused often and need to set many parameters, but use of standard PostScript operators is normally efficient enough.

Because graphics states occupy a fair amount of memory, they can actually degrade performance. Be sure to test application performance with and without the private graphics state before committing to its use.

See also: – **setUpGState**, – **gState**, – **renewGState**, – **releaseGState,** – **lockFocus**

ancestorSharedWithView:

– (NSView *)**ancestorSharedWithView:**(NSView *)*aView*

Returns the closest ancestor shared by the receiver and *aView*, or **nil** if there's no such object. Returns **self** if *aView* is identical to the receiver.

See also: – **isDescendantOf:**

autoresizesSubviews

– (BOOL)**autoresizesSubviews**

Returns YES if the receiver automatically resizes its subviews using **resizeSubviewsWithOldSize:** whenever its frame size changes, NO otherwise.

See also: – **setAutoresizesSubviews:**

autoresizingMask

– (unsigned int)**autoresizingMask**

Returns the receiver's autoresizing mask, which determines how it's resized by the **resizeWithOldSuperviewSize:** method. The autoresizing mask values are listed under the **setAutoresizingMask:** method description. If the autoresizing mask is equal to NSViewNotSizable (that is, if none of the options are set), then the receiver doesn't resize at all in **resizeWithOldSuperviewSize:**.

autoscroll:

– (BOOL)**autoscroll:**(NSEvent *)*theEvent*

Scrolls the receiver's closest ancestor NSClipView proportionally to *theEvent*'s distance outside of it. *theEvent*'s location should be expressed in the window's base coordinate system (which it normally is), not the receiving view object's. Returns YES if any scrolling is performed; otherwise returns NO.

View objects that track mouse-dragged events can use this method to scroll automatically when the mouse is dragged outside of the NSClipView. Repeated invocations of this method (with an appropriate delay) result in continual scrolling, even when the mouse doesn't move.

See also: – **autoscroll:** (NSClipView), – **scrollPoint:**, – **isDescendantOf:**

beginPage:label:bBox:fonts:

– (void)**beginPage:**(int)*ordinalNum*
label:(NSString *)*aString*
bBox:(NSRect)*pageRect*
fonts:(NSString *)*fontNames*

Writes a conforming PostScript page separator. This method is invoked by **print:** and **fax:**.

ordinalNum is the page's position in the document's page sequence (from 1 through *n* for an *n*-page document).

aString is a string that contains no white space characters. It identifies the page according to the document's internal numbering scheme. If *aString* is empty (@""), the text equivalent of *ordinalNum* is used.

pageRect is the rectangle enclosing all the drawing on the page about to be printed, in the default PostScript coordinate system of the page (not of the receiving NSView). If *pageRect* is an empty rectangle (width and height of zero), "(atend)" is output instead of a description of the bounding box, and the bounding box is output at the end of the page.

fontNames is a string containing the names of the fonts used in the page, each pair separated by a space. If the fonts used are unknown before the page is printed, *fontNames* can be empty. In this case "(atend)" is output instead of the font names, which are listed automatically at the end of the page description.

See also: – **endPage**, **NSIsEmptyRect()** (Foundation Kit)

beginPageSetupRect:placement:

– (void)**beginPageSetupRect:**(NSRect)*aRect* **placement:**(NSPoint)*location*

Writes the page setup section for a page, generating the initial coordinate transformation for printing the region defined by *aRect* in the receiver's coordinate system. *location* is the offset in page coordinates of the rectangle on the physical page.

This method is invoked by **print:** and **fax:** after the starting comments for the page have been written. It generates a PostScript **save** operation and invokes **lockFocus**, which are balanced in the **endPage** method with an **unlockFocus** and a PostScript **restore** operation.

See also: – **addToPageSetup**

beginPrologueBBox:creationDate:createdBy:fonts:forWhom:pages:title:

– (void)**beginPrologueBBox:**(NSRect)*boundingBox*
creationDate:(NSString *)*dateCreated*
createdBy:(NSString *)*anApplication*
fonts:(NSString *)*fontNames*
forWhom:(NSString *)*user*
pages:(int)*numPages*
title:(NSString *)*aTitle*

Invoked by **print:** and **fax:** to write the start of a conforming PostScript header.

boundingBox is the bounding box of the document, expressed in the default PostScript coordinate system on the page. The document bounding box is the union of the bounding boxes of every page in the document. If it's unknown, *boundingBox* should be empty (width and height of zero). In this case "(atend)" is output instead of the bounding box, which is accumulated as pages are printed and written in the trailer.

dateCreated is a text string containing a human readable date. If *dateCreated* is empty (@"") the current date is used.

anApplication is a string containing the name of the document creator. If *anApplication* is empty then the string returned by NSProcessInfo's **processName** instance method is used.

fontNames is a string holding the names of the fonts used in the document, each pair separated by a space. If the fonts used are unknown before the document is printed, *fontNames* should be empty. In this case "(atend)" is output instead of the font names, and the name of each NSFont used by the view is written in the trailer.

user is a string containing the name of the person the document is being printed for. If *user* is empty the login name of the current user is substituted.

numPages specifies the number of pages in the document. If unknown at the beginning of printing, *numPages* should have a value of –1. In this case "(atend)" is output instead of a page count, the pages are counted as they are generated, and the resulting count is written in the trailer.

aTitle is a string specifying the title of the document. If *aTitle* is empty, then the title of the receiver's window is used. If the window has no title, "Untitled" is output.

See also: – **beginTrailer**, – **endTrailer**, – **set** (NSFont), + **useFont:** (NSFont)

beginSetup

– (void)**beginSetup**

Writes the beginning of the document setup section, which begins with a %%BeginSetup comment and includes a %%PaperSize comment declaring the type of paper being used. This method is invoked by **print:** and **fax:** at the start of the setup section of the document, which occurs after the prologue of the document

has been written, but before any pages are written. This section of the output is intended for device setup or general initialization code.

beginTrailer

– (void)**beginTrailer**

Writes the start of a conforming PostScript trailer, which begins with a %%Trailer comment. This method is invoked by **print:** and **fax:** immediately after all pages have been written.

bounds

– (NSRect)**bounds**

Returns the receiver's bounds rectangle, which expresses its location and size in its own coordinate system. The bounds rectangle may be rotated; use the **boundsRotation** method to check this.

See also: – **frame**, – **setBounds:**

boundsRotation

– (float)**boundsRotation**

Returns the angle of the receiver's bounds rectangle relative to its frame rectangle. See the **setBoundsRotation:** method description for more information on bounds rotation.

See also: – **rotateByAngle:**, – **setBoundsRotation:**

canDraw

– (BOOL)**canDraw**

Returns YES if drawing commands will produce any result, NO otherwise. Use this method when invoking a draw method directly along with **lockFocus** and **unlockFocus**, bypassing the **display...** methods (which test drawing ability and perform locking for you). If this method returns NO, you shouldn't invoke **lockFocus** or perform any drawing.

An NSView can draw if it's attached to a view hierarchy in an NSWindow and the NSWindow has a corresponding PostScript window device, or during printing if the NSView is a descendant of the view being printed.

centerScanRect:

– (NSRect)**centerScanRect:**(NSRect)*aRect*

Converts the corners of a rectangle to lie on the center of device pixels, which is useful in compensating for PostScript overscanning when the coordinate system has been scaled. This method converts the given rectangle to device coordinates, adjusts the rectangle to lie in the center of the pixels, and converts the resulting rectangle back to the receiver's coordinate system. Returns the adjusted rectangle.

See also: – **isRotatedOrScaledFromBase**

convertPoint:fromView:

– (NSPoint)**convertPoint:**(NSPoint)*aPoint* **fromView:**(NSView *)*aView*

Converts *aPoint* from *aView*'s coordinate system to that of the receiver. If *aView* is **nil**, this method instead converts from window base coordinates. Both *aView* and the receiver must belong to the same NSWindow. Returns the converted point.

See also: – **convertRect:fromView:**, – **convertSize:fromView:**, – **ancestorSharedWithView:**, – **contentView** (NSWindow)

convertPoint:toView:

– (NSPoint)**convertPoint:**(NSPoint)*aPoint* **toView:**(NSView *)*aView*

Converts *aPoint* from the receiver's coordinate system to that of *aView*. If *aView* is **nil**, this method instead converts to window base coordinates. Both *aView* and the receiver must belong to the same NSWindow. Returns the converted point.

See also: – **convertRect:toView:**, – **convertSize:toView:**, – **ancestorSharedWithView:**, – **contentView** (NSWindow)

convertRect:fromView:

– (NSRect)**convertRect:**(NSRect)*aRect* **fromView:**(NSView *)*aView*

Converts *aRect* from *aView*'s coordinate system to that of the receiver. If *aView* is **nil**, this method instead converts from window base coordinates. Both *aView* and the receiver must belong to the same NSWindow. Returns the converted rectangle.

See also: – **convertPoint:fromView:**, – **convertSize:fromView:**, – **ancestorSharedWithView:**, – **contentView** (NSWindow)

convertRect:toView:

– (NSRect)**convertRect:**(NSRect)*aRect* **toView:**(NSView *)*aView*

Converts *aRect* from the receiver's coordinate system to that of *aView*. If *aView* is **nil**, this method instead converts to window base coordinates. Both *aView* and the receiver must belong to the same NSWindow. Returns the converted rectangle.

See also: – **convertPoint:toView:**, – **convertSize:toView:**, – **ancestorSharedWithView:**, – **contentView** (NSWindow)

convertSize:fromView:

– (NSSize)**convertSize:**(NSSize)*aSize* **fromView:**(NSView *)*aView*

Converts *aSize* from *aView*'s coordinate system to that of the receiver. If *aView* is **nil**, this method instead converts from window base coordinates. Both *aView* and the receiver must belong to the same NSWindow. Returns the converted size.

See also: – **convertPoint:fromView:**, – **convertRect:fromView:**, – **ancestorSharedWithView:**, – **contentView** (NSWindow)

convertSize:toView:

– (NSSize)**convertSize:**(NSSize)*aSize* **toView:**(NSView *)*aView*

Converts *aSize* from the receiver's coordinate system to that of *aView*. If *aView* is **nil**, this method instead converts to window base coordinates. Both *aView* and the receiver must belong to the same NSWindow. Returns the converted size.

See also: – **convertPoint:toView:**, – **convertRect:toView:**, – **ancestorSharedWithView:**, – **contentView** (NSWindow)

dataWithEPSInsideRect:

– (NSData *)**dataWithEPSInsideRect:**(NSRect)*aRect*

Returns EPS data that draws the region of the receiver within *aRect*. This data can be placed on an NSPasteboard, written to a file, or used to create an NSImage object.

See also: – **writeEPSInsideRect:toPasteboard:**

discardCursorRects

– (void)**discardCursorRects**

Invalidates all cursor rectangles set up using **addCursorRect:cursor:**. You need never invoke this method directly; it's invoked automatically before the NSView's cursor rectangles are reestablished using **resetCursorRects**.

See also: – **discardCursorRects** (NSWindow)

display

– (void)**display**

Displays the receiver and all its subviews if possible, invoking each NSView's **lockFocus**, **drawRect:**, and **unlockFocus** methods as necessary. If the receiver isn't opaque, this method backs up the view hierarchy to the first opaque ancestor, calculates the portion of the opaque ancestor covered by the receiver, and begins displaying from there.

See also: – **canDraw**, – **opaqueAncestor**, – **visibleRect**, – **displayIfNeededIgnoringOpacity**

displayIfNeeded

– (void)**displayIfNeeded**

Displays the receiver and all its subviews if any part of the receiver has been marked as needing display with a **setNeedsDisplay:** or **setNeedsDisplayInRect:** message. This method invokes each NSView's **lockFocus**, **drawRect:**, and **unlockFocus** methods as necessary. If the receiver isn't opaque, this method backs up the view hierarchy to the first opaque ancestor, calculates the portion of the opaque ancestor covered by the receiver, and begins displaying from there.

See also: – **display**, – **needsDisplay**, – **displayIfNeededIgnoringOpacity**

displayIfNeededIgnoringOpacity

– (void)**displayIfNeededIgnoringOpacity**

Acts as **displayIfNeeded**, except that this method doesn't back up to the first opaque ancestor—it simply causes the receiver and its descendants to execute their drawing code.

displayIfNeededInRect:

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

Acts as **displayIfNeeded**, confining drawing to *aRect*.

displayIfNeededInRectIgnoringOpacity:

– (void)**displayIfNeededInRectIgnoringOpacity:**(NSRect)*rect*

Acts as **displayIfNeeded**, but confining drawing to *aRect* and not backing up to the first opaque ancestor—it simply causes the receiver and its descendants to execute their drawing code.

displayRect:

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

Acts as **display**, confining drawing to *aRect*.

displayRectIgnoringOpacity:

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

Acts as **display**, but confining drawing to *aRect* and not backing up to the first opaque ancestor—it simply causes the receiver and its descendants to execute their drawing code.

dragFile:fromRect:slideBack:event:

– (BOOL)**dragFile:**(NSString *)*fullPath*
fromRect:(NSRect)*aRect*
slideBack:(BOOL)*flag*
event:(NSEvent *)*theEvent*

Initiates a dragging operation from the receiver, allowing the user to drag a file icon to any application that has window or view objects that accept files. This method must be invoked only within an implementation of the **mouseDown:** method. Returns YES if the receiver successfully initiates the dragging operation (which doesn't necessarily mean the dragging operation concluded successfully). Otherwise returns NO.

The dragging operation uses these arguments:

- *fullPath* is an absolute path for the file to be dragged.
- *aRect* describes the position of the icon in the receiver's coordinate system.
- *flag* indicates whether the icon being dragged should slide back to its position in the receiver if the file isn't accepted. The icon slides back to *aRect*, if *flag* is YES, the file is not accepted by the dragging destination, and the user has not disabled icon animation; otherwise it simply disappears.
- *theEvent* is the mouse-down event object from which to initiate the drag operation. In particular, its mouse location is used for the offset of the icon being dragged.

See the NSDraggingSource, NSDraggingInfo, and NSDraggingDestination protocol specifications for more information on dragging operations.

See also: – **dragImage:at:offset:event:pasteboard:source:slideBack:**,
– **shouldDelayWindowOrderingForEvent:**

dragImage:at:offset:event:pasteboard:source:slideBack:

– (void)**dragImage:**(NSImage *)*anImage*
at:(NSPoint)*imageLoc*
offset:(NSSize)*mouseOffset*
event:(NSEvent *)*theEvent*
pasteboard:(NSPasteboard *)*pboard*
source:(id)*sourceObject*
slideBack:(BOOL)*flag*

Initiates a dragging operation from the receiver, allowing the user to drag arbitrary data with a specified icon into any application that has window or view objects that accept dragged data. This method must be invoked only within an implementation of the **mouseDown:** method. The dragging operation uses these arguments:

- *anImage* is the NSImage to be dragged.
- *imageLoc* is the location of the image's lower left corner, in the receiver's coordinate system. It determines the placement of the dragged image under the cursor.
- *mouseOffset* is the mouse's current location relative to the mouse-down location. It determines the initial location of the image when dragging commences. If you initiate a dragging operation immediately on a mouse-down event, this should be (0.0, 0.0). If you test for a mouse-dragged event first, this should be the difference between the mouse-dragged event's location and that of the mouse-down event.
- *theEvent* is the left-mouse-down event that triggered the dragging operation (see below).
- *pboard* holds the data to be transferred to the destination (see below).
- *sourceObject* serves as the controller of the dragging operation. It must conform to the NSDraggingSource protocol, and is typically the receiver itself or its NSWindow.
- *flag* determines whether the NSImage should slide back if it's rejected. The image slides back to *aPoint* if *flag* is YES, the image isn't accepted by the dragging destination, and the user hasn't disabled icon animation; otherwise it simply disappears.

Before invoking this method, you must place the data to be transferred on *pboard*. To do this, get the drag pasteboard object (NSDragPboard), declare the types of the data, and then put the data on the pasteboard. This code fragment initiates a dragging operation on an image itself (that is, the image is the data to be transferred):

```
- (void)mouseDown:(NSEvent *)theEvent
{
    NSSize dragOffset = NSMakeSize(0.0, 0.0);
    NSPasteboard *pboard;

    pboard = [NSPasteboard pasteboardWithName:NSDragPboard];
    [pboard declareTypes:[NSArray arrayWithObject:NSTIFFPboardType] owner:self];
    [pboard setData:[[self image] TIFFRepresentation] forType:NSTIFFPboardType];

    [self dragImage:[self image] at:[self imageLocation] offset:dragOffset
        event:theEvent pasteboard:pboard source:self slideBack:YES];

    return;
}
```

See the NSDraggingSource, NSDraggingInfo, and NSDraggingDestination protocol specifications for more information on dragging operations.

See also: – **dragFile:fromRect:slideBack:event:**, – **shouldDelayWindowOrderingForEvent:**

drawPageBorderWithSize:

– (void)**drawPageBorderWithSize:**(NSSize)*borderSize*

Allows applications that use the Application Kit pagination facility to draw additional marks on each logical page, such as alignment marks or a virtual sheet border. This method is invoked by **beginPageSetupRect: placement:**. The default implementation doesn't draw anything.

See also: – **drawSheetBorderWithSize:**

drawRect:

– (void)**drawRect:**(NSRect)*aRect*

Overridden by subclasses to draw the receiver's image within *aRect*. The receiver can assume that the PostScript focus has been locked, that drawing will be clipped to its frame rectangle, and that the coordinate transformations of its frame and bounds rectangles have been applied; all it need do is invoke PostScript client functions. *aRect* is provided for optimization; it's perfectly correct, though inefficient, to draw images that lie outside the requested rectangle. See "How to Draw" in the class description for information and references on drawing.

This method is intended to be completely overridden by each subclass that performs drawing. Don't invoke **super**'s implementation in your subclass.

See also: – **display**, – **shouldDrawColor**, – **isFlipped**

drawSheetBorderWithSize:

– (void)**drawSheetBorderWithSize:**(NSSize)*borderSize*

Allows applications that use the Application Kit pagination facility to draw additional marks on each printed sheet, such as crop marks or fold lines. This method is invoked by **beginPageSetupRect: placement:**. The default implementation doesn't draw anything.

See also: – **drawPageBorderWithSize:**

enclosingScrollView

– (NSScrollView *)**enclosingScrollView**

Returns the nearest ancestor NSScrollView containing the receiver (*not* including the receiver itself); otherwise returns **nil**.

endHeaderComments

– (void)**endHeaderComments**

Writes out the end of a conforming PostScript header, starting with the %%EndComments line and then the start of the prologue, including the Application Kit's standard printing package. Override **endPrologue** to add your own global definitions. This method is invoked by **print:** and **fax:** after **beginPrologueBBox: creationDate:createdBy:fonts:forWhom:pages:title:** and before **endPrologue**.

endPage

– (void)**endPage**

Writes the end of a conforming PostScript page. This method is invoked after each page is printed. It balances the preceding invocation of **beginPageSetupRect:placement:** by invoking **unlockFocus** and generating a PostScript **restore** operator, and generates a PostScript **showpage** operator to finish the page. This method also generates comments for the bounding box and page fonts, if they were specified as being at the end of the page.

See also: – **beginPrologueBBox:creationDate:createdBy:fonts:forWhom:pages:title:**

endPageSetup

– (void)**endPageSetup**

Writes the end of the page setup section, which begins with a %%EndPageSetup comment. This method is invoked by **print:** and **fax:** just after **beginPageSetupRect:placement:** is invoked.

endPrologue

– (void)**endPrologue**

Writes the end of the conforming PostScript prologue. This method is invoked by **print:** and **fax:** after the prologue of the document has been written. Subclasses can override this method to add their own definitions to the prologue. For example:

```
- endPrologue
{
    [[NSDPSContext currentContext] printFormat:@"/littleProc {pop} def");
    [super endPrologue];
    return;
}
```

endSetup

– (void)**endSetup**

Writes out the end of the setup section, which begins with a %%EndSetup comment. This method is invoked by **print:** and **fax:** just after **beginSetup** is invoked.

endTrailer

– (void)**endTrailer**

Writes the end of the conforming PostScript trailer. This method is invoked by **print:** and **fax:** just after **beginTrailer** is invoked.

See also: – **beginTrailer**

fax:

– (void)**fax:**(id)*sender*

Opens the Fax panel, and if the user chooses an option other than canceling, prints the receiver and all its subviews to a fax modem.

See also: – **print:**

frame

– (NSRect)**frame**

Returns the receiver's frame rectangle, which defines its position in its superview. The frame rectangle may be rotated; use the **frameRotation** method to check this.

See also: – **bounds**, – **setFrame:**

frameRotation

– (float)**frameRotation**

Returns the angle of the receiver's frame relative to its superview's coordinate system.

See also: – **setFrameRotation:**, – **boundsRotation**

gState

– (int)**gState**

Returns the PostScript user object identifier for the receiver's PostScript graphics state object, or 0 if the receiver doesn't have a graphics state object. A view object's graphics state object is recreated from scratch whenever the view is focused, unless the **allocateGState** method has been invoked. So if the receiver hasn't been focused or hasn't received the **allocateGState** message, this method returns 0. For more information on graphics objects and when they are created, see the "PostScript Graphics State Objects" section.

Although applications rarely need to use the value returned by **gState**, it can be passed to the few PostScript operators that take an object identifier as a parameter, such as **PScomposite** and **PSdissolve**.

See also: – **allocateGState**, – **setUpGState**, – **renewGState**, – **releaseGState**, – **lockFocus**

heightAdjustLimit

– (float)**heightAdjustLimit**

Returns the fraction (between 0.0 and 1.0) of the page that can be pushed onto the next page during automatic pagination to prevent items such as lines of text from being divided across pages. This fraction is used to calculate the bottom edge limit for a **adjustPageHeightNew:top:bottom:limit:** message.

See also: – **widthAdjustLimit**

hitTest:

– (NSView *)**hitTest:**(NSPoint)*aPoint*

Returns the farthest descendant of the receiver in the view hierarchy (including itself) that contains *aPoint*, or **nil** if *aPoint* lies completely outside the receiver. *aPoint* is in the coordinate system of the receiver's superview, not of the receiver itself.

This method is used primarily by an NSWindow to determine which NSView should receive a mouse-down event. You'd rarely need invoke this method, but you might want to override it to have a view object hide mouse-down events from its subviews.

See also: – **mouse:inRect:**, – **convertPoint:toView:**

initWithFrame:

– (id)**initWithFrame:**(NSRect)*frameRect*

Initializes a newly allocated NSView with *frameRect* as its frame rectangle. The new view object must be inserted into the view hierarchy of an NSWindow before it can be used. This method is the designated initializer for the NSView class. Returns **self**.

See also: – **addSubview:**, – **addSubview:positioned:relativeTo:**, – **setFrame:**

interfaceStyle

– (NSInterfaceStyle)**interfaceStyle**

Returns the receiver's interface style, such as NSMacintoshInterfaceStyle or NSWindows95InterfaceStyle. A responder's style (if other than NSNoInterfaceStyle) overrides all other settings, such as those established by the defaults system.

See also: – **setInterfaceStyle:**

isDescendantOf:

– (BOOL)**isDescendantOf:**(NSView *)*aView*

Returns YES if the receiver is a subview, immediate or not, of *aView*, or if it's identical to *aView*; otherwise returns NO.

See also: – **superview**, – **subviews**, – **ancestorSharedWithView:**

isFlipped

– (BOOL)**isFlipped**

Returns YES if the receiver uses flipped drawing coordinates or NO if it uses native PostScript coordinates. NSView's implementation returns NO; subclasses that use flipped coordinates should override this method to return YES.

isOpaque

– (BOOL)**isOpaque**

Overridden by subclasses to return YES if the receiver is opaque, NO otherwise. A view object is opaque if it completely covers its frame rectangle when drawing itself. NSView, being an abstract class, performs no drawing at all and so returns NO.

See also: – **opaqueAncestor**, – **displayRectIgnoringOpacity:**, – **displayIfNeededIgnoringOpacity**, – **displayIfNeededInRectIgnoringOpacity:**

isRotatedFromBase

– (BOOL)**isRotatedFromBase**

Returns YES if the receiver or any of its ancestors has ever received a **setFrameRotation:** or **setBoundsRotation:** message; otherwise returns NO. This intent of this information is to optimize drawing and coordinate calculation, not necessarily to reflect the exact state of the receiver's coordinate system, so it may not reflect the actual rotation. For example, if an NSView is rotated to 45 degrees and later back to zero, this method still returns YES.

See also: – **frameRotation**, – **boundsRotation**

isRotatedOrScaledFromBase

– (BOOL)**isRotatedOrScaledFromBase**

Returns YES if the receiver or any of its ancestors have ever had a nonzero frame or bounds rotation, or has been scaled from the window's base coordinate system; otherwise returns NO. This intent of this information is to optimize drawing and coordinate calculation, not necessarily to reflect the exact state of the receiver's coordinate system, so it may not reflect the actual rotation or scaling. For example, if an NSView is rotated to 45 degrees and later back to zero, this method still returns YES.

See also: – **frameRotation**, – **boundsRotation**, – **centerScanRect:**, – **setBounds:**, – **setBoundsSize:**, – **scaleUnitSquareToSize:**

knowsPagesFirst:last:

– (BOOL)**knowsPagesFirst:**(int *)*firstPageNum* **last:**(int *)*lastPageNum*

Overridden by subclasses to indicate whether the receiver wishes to perform its own pagination. This method is invoked by **print:** and **fax:**. If the receiver returns NO, it's paginated by NSView's automatic pagination mechanism. If the receiver returns YES, the printing mechanism later invokes **rectForPage:** to determine the rectangle of each page from the out parameters *firstPageNum* to *lastPageNum*. NSView's implementation returns NO.

This method is normally invoked with the value of *firstPageNum* set to 1 and of *lastPageNum* set to the maximum integer size. If the receiver returns YES it must alter these values to reflect its own numbering scheme, and possibly to limit which pages are printed.

See also: – **rectForPage:**

locationOfPrintRect:

– (NSPoint)**locationOfPrintRect:**(NSRect)*aRect*

Invoked by **print:** and **fax:** to determine the location of *aRect*, the rectangle being printed on the physical page. The return value of this method is used to set the origin for *aRect*, whose size the receiver can examine in order to properly place it. Both the rectangle and the returned location are expressed in the default PostScript coordinate system of the page.

NSView's implementation places *aRect* according to the status of the NSPrintInfo object for the print job. By default it places the image in the upper left corner of the page, but if NSPrintInfo's **isHorizontallyCentered** or **isVerticallyCentered** method returns YES, it centers a single-page image along the appropriate axis. A multiple-page document, however, is always placed so that the divided pieces can be assembled at their edges.

lockFocus

– (void)**lockFocus**

Locks the PostScript focus on the receiver, so that subsequent PostScript commands take effect in the receiver's window and coordinate system. If you don't use a **display...** method to draw an NSView, you must invoke **lockFocus** before invoking methods that send PostScript commands to the Window Server, and must balance it with an **unlockFocus** message when finished.

See also: + **focusView**, – **display**, – **drawRect:**

menuForEvent:

– (NSMenu *)**menuForEvent:**(NSEvent *)*theEvent*

Overridden by subclasses to return a context-sensitive pop-up menu for the mouse-up event *theEvent*. The receiver can use information in the mouse event, such as its location over a particular element of the receiver, to determine what kind of menu to return. For example, a text object might display a text-editing menu when the mouse lies over text and a menu for changing graphic attributes when the mouse lies over an embedded image.

NSView's implementation returns the receiver's normal menu. This menu is used only on Microsoft Windows.

See also: + **defaultMenu**, – **menu** (NSResponder)

mouse:inRect:

– (BOOL)**mouse:**(NSPoint)*aPoint* **inRect:**(NSRect)*aRect*

Returns YES if *aRect* contains *aPoint* (which represents the hot spot of the mouse cursor), **accounting for whether the receiver is flipped or not.** *aPoint* and *aRect* must be expressed in the receiver's coordinate system.

Never use the Foundation Kit's **NSPointInRect()** function as a substitute for this method. It doesn't account for flipped coordinate systems.

See also: – **hitTest:**, – **isFlipped**, **NSMouseInRect()** (Foundation Kit), – **convertPoint:fromView:**

needsDisplay

– (BOOL)**needsDisplay**

Returns YES if the receiver needs to be displayed, as indicated using the **setNeedsDisplay:** and **setNeedsDisplayInRect:** methods; returns NO otherwise. The **displayIfNeeded...** methods check this status to avoid unnecessary drawing, and all display methods clear this status to indicate that the view object is up to date.

needsPanelToBecomeKey

– (BOOL)**needsPanelToBecomeKey**

Overridden by subclasses to return YES if the receiver requires its panel, which might otherwise avoid becoming key, to become the key window so that it can handle keyboard input. Such a subclass should also override **acceptsFirstResponder** to return YES. NSView's implementation returns NO.

See also: – **becomesKeyOnlyIfNeeded** (NSPanel)

nextKeyView

– (NSView *)**nextKeyView**

Returns the view object following the receiver in the key view loop, or **nil** if there is none. This view should, if possible, be made first responder when the user navigates forward from the receiver using keyboard interface control.

See also: – **nextValidKeyView**, – **setNextKeyView:**, – **previousKeyView**, – **previousValidKeyView**

nextValidKeyView

– (NSView *)**nextValidKeyView**

Returns the closest view object in the key view loop that follows the receiver and actually accepts first responder status, or **nil** if there is none.

See also: – **nextKeyView**, – **setNextKeyView:**, – **previousKeyView**, – **previousValidKeyView**

opaqueAncestor

– (NSView *)**opaqueAncestor**

Returns the receiver's closest opaque ancestor (including the receiver itself).

See also: – **isOpaque**, – **displayRectIgnoringOpacity:**, – **displayIfNeededIgnoringOpacity**, – **displayIfNeededInRectIgnoringOpacity:**

performKeyEquivalent:

– (BOOL)**performKeyEquivalent:**(NSEvent *)*theEvent*

Implemented by subclasses to respond to key equivalents (also known as shortcuts). If the receiver's key equivalent is the same as the characters of the key-down event *theEvent*, as returned by **charactersIgnoringModifiers**, it should take the appropriate action and return YES. Otherwise, it should return the result invoking **super**'s implementation. NSView's implementation of this method simply passes the message down the view hierarchy (from superviews to subviews) and returns NO if none of the receiver's subviews responds YES.

See also: – **performMnemonic:**, – **keyDown:** (NSWindow)

performMnemonic:

– (BOOL)**performMnemonic:**(NSString *)*aString*

Implemented by subclasses to respond to mnemonics. If the receiver's mnemonic is the same as the characters of the key-down event *theEvent*, as returned by **charactersIgnoringModifiers**, it should take the appropriate action and return YES. Otherwise, it should return the result invoking **super**'s implementation. NSView's implementation of this method simply passes the message down the view hierarchy (from superviews to subviews) and returns NO if none of the receiver's subviews responds YES.

See also: – **performKeyEquivalent:**, – **keyDown:** (NSWindow)

postsBoundsChangedNotifications

– (BOOL)**postsBoundsChangedNotifications**

Returns YES if the receiver posts notifications to the default notification center whenever its bounds rectangle changes; returns NO otherwise. See **setPostsBoundsChangedNotifications:** for a list of methods that result in notifications.

postsFrameChangedNotifications

– (BOOL)**postsFrameChangedNotifications**

Returns YES if the receiver posts notifications to the default notification center whenever its frame rectangle changes; returns NO otherwise. See **setFrameRotation:** for a list of methods that result in notifications.

previousKeyView

– (NSView *)**previousKeyView**

Returns the view object preceding the receiver in the key view loop, or **nil** if there is none. This view should, if possible, be made first responder when the user navigates backward from the receiver using keyboard interface control.

See also: – **previousValidKeyView**, – **nextKeyView**, – **nextValidKeyView**, – **setNextKeyView:**

previousValidKeyView

– (NSView *)**previousValidKeyView**

Returns the closest view object in the key view loop that precedes the receiver and actually accepts first responder status, or **nil** if there is none.

See also: – **previousKeyView**, – **nextValidKeyView**, – **nextKeyView**, – **setNextKeyView:**

print:

– (void)**print:**(id)*sender*

Opens the Print panel, and if the user chooses an option other than canceling, prints the receiver and all its subviews to the device specified in the Print panel.

See also: – **fax:**, – **dataWithEPSInsideRect:**, – **writeEPSInsideRect:toPasteboard:**

rectForPage:

– (NSRect)**rectForPage:**(int)*pageNumber*

Implemented by subclasses to determine the portion of the receiver to be printed for page number *page*. If the receiver responded YES to an earlier **knowsPagesFirst:last:** message, this method is invoked for each page it specified in the out parameters of that message. The receiver is later made to display this rectangle in order to generate the image for this page. This method should return NSZeroRect if *pageNumber* is outside the receiver's bounds.

If an NSView responds NO to **knowsPagesFirst:last:**, this method isn't invoked by the printing mechanism.

See also: – **adjustPageHeightNew:top:bottom:limit:**, – **adjustPageWidthNew:left:right:limit:**

reflectScrolledClipView:

– (void)**reflectScrolledClipView:**(NSClipView *)*aClipView*

Notifies *aClipView*'s superview that either *aClipView*'s bounds rectangle or the document view's frame rectangle has changed, and that any indicators of the scroll position need to be adjusted. NSScrollView implements this method to update its NSScrollers.

registerForDraggedTypes:

– (void)**registerForDraggedTypes:**(NSArray *)*pboardTypes*

Registers *pboardTypes* as the pasteboard types that the receiver will accept as the destination of an image-dragging session.

Note: Registering an NSView for dragged types automatically makes it a candidate destination object for a dragging session. As such, it must properly implement some or all of the NSDraggingDestination protocol methods. As a convenience, NSView provides default implementations of these methods. See the NSDraggingDestination protocol specification for details.

See also: – **unregisterDraggedTypes**

releaseGState

– (void)**releaseGState**

Frees the receiver's PostScript graphics state object, if it has one.

See also: – **allocateGState**

removeCursorRect:cursor:

– (void)**removeCursorRect:**(NSRect)*aRect* **cursor:**(NSCursor *)*aCursor*

Completely removes a cursor rectangle from the receiver. *aRect* and *aCursor* must match values previously specified using **addCursorRect:cursor:**.

You should rarely need to use this method. **resetCursorRects**, which is invoked any time cursor rectangles need to be rebuilt, should establish only the cursor rectangles needed. If you implement **resetCursorRects** in this way, you can then simply modify the state that **resetCursorRects** uses to build its cursor rectangles and then invoke NSWindow's **invalidateCursorRectsForView:**.

See also: – **discardCursorRects**

removeFromSuperview

– (void)**removeFromSuperview**

Unlinks the receiver from its superview and its NSWindow, removes it from the responder chain, and invalidates its cursor rectangles. The receiver is also released; if you plan to reuse it, be sure to retain it before sending this message and to release it as appropriate when adding it as a subview of another NSView.

Never invoke this method during display.

See also: – **addSubview:**, – **addSubview:positioned:relativeTo:**,
– **removeFromSuperviewWithoutNeedingDisplay**

removeFromSuperviewWithoutNeedingDisplay

– (void)**removeFromSuperviewWithoutNeedingDisplay**

Unlinks the receiver from its superview and its NSWindow, removes it from the responder chain, but *does not* invalidate its cursor rectangles to cause redrawing. The receiver is also released; if you plan to reuse it, be sure to retain it before sending this message and to release it as appropriate when adding it as a subview of another NSView.

Unlike its counterpart, **removeFromSuperview**, this method can be safely invoked during display.

See also: – **addSubview:**, – **addSubview:positioned:relativeTo:**

removeTrackingRect:

– (void)**removeTrackingRect:**(NSTrackingRectTag)*aTag*

Removes the tracking rectangle identified by *aTag*, which is the value returned by a previous **addTrackingRect:owner:userData:assumeInside:** message.

renewGState

– (void)**renewGState**

Invalidates the receiver's PostScript graphics state object, if it has one, so that it will be regenerated using **setUpGState** the next time the receiver is focused for drawing.

See also: – **lockFocus**

replaceSubview:with:

– (void)**replaceSubview:**(NSView *)*oldView* **with:**(NSView *)*newView*

Replaces *oldView* with *newView* in the receiver's subviews. Does nothing and returns **nil** if *oldView* is not a subview of the receiver.

This method causes *oldView* to be released; if you plan to reuse it, be sure to retain it before sending this message and to release it as appropriate when adding it as a subview of another NSView.

See also: – **addSubview:**, **– addSubview:positioned:relativeTo:**

resetCursorRects

– (void)**resetCursorRects**

Overridden by subclasses to define their default cursor rectangles. A subclass's implementation must invoke **addCursorRect:cursor:** for each cursor rectangle it wants to establish. NSView's implementation does nothing.

Application code should never invoke this method directly; it's invoked automatically as described in the "Tracking Rectangles and Cursor Rectangles" section. Use the **invalidateCursorRectsForView:** method instead to explicitly rebuild cursor rectangles.

See also: – **visibleRect**

resizeSubviewsWithOldSize:

– (void)**resizeSubviewsWithOldSize:**(NSSize)*oldFrameSize*

Informs the receivers's subviews that the receiver's bounds rectangle size has changed from *oldFrameSize*. If the receiver is configured to autoresize its subviews, this method is automatically invoked by any method that changes the receiver's frame size.

NSView's implementation sends **resizeWithOldSuperviewSize:** to the receiver's subviews with *oldFrameSize* as the argument. You shouldn't invoke this method directly, but you can override it to define a specific retiling behavior.

See also: – **setAutoresizesSubviews:**

resizeWithOldSuperviewSize:

– (void)**resizeWithOldSuperviewSize:**(NSSize)*oldFrameSize*

Informs the receiver that the frame size of its superview has changed from *oldFrameSize*. This method is normally invoked automatically from **resizeSubviewsWithOldSize:**.

NSView's implementation resizes the receiver according to the autoresizing options listed under the **setAutoresizingMask:** method description. You shouldn't invoke this method directly, but you can override it to define a specific resizing behavior.

rotateByAngle:

– (void)**rotateByAngle:**(float)*angle*

Rotates the receiver's bounds rectangle by *angle* degrees around the origin of the coordinate system, (0.0, 0.0) See the **setBoundsRotation:** method description for more information.This method neither redisplays the receiver nor marks it as needing display. You must do this yourself with **display** or **setNeedsDisplay:**.

This method posts an NSViewBoundsDidChangeNotification to the default notification center if the receiver is configured to do so.

See also: – **setFrameRotation:**, – **setPostsBoundsChangedNotifications:**

scaleUnitSquareToSize:

– (void)**scaleUnitSquareToSize:**(NSSize)*newUnitSize*

Scales the receiver's coordinate system so that the unit square changes to *newUnitSize*. For example, a *newUnitSize* of (0.5, 1.0) causes the receiver's horizontal coordinates to be halved, in turn doubling the width of its bounds rectangle. Note that scaling is performed from the origin of the coordinate system, (0.0,

0.0), not the origin of the bounds rectangle; as a result, both the origin and size of the bounds rectangle are changed. The frame rectangle remains unchanged.

This method neither redisplays the receiver nor marks it as needing display. You must do this yourself with **display** or **setNeedsDisplay:**.

This method posts an NSViewBoundsDidChangeNotification to the default notification center if the receiver is configured to do so.

See also: – **setBoundsSize:**, – **setPostsBoundsChangedNotifications:**

scrollClipView:toPoint:

– (void)**scrollClipView:**(NSClipView *)*aClipView* **toPoint:**(NSPoint)*newOrigin*

Notifies *aClipView*'s superview that *aClipView* needs to set its bounds rectangle origin to *newOrigin*. *aClipView*'s superview should then send a **scrollToPoint:** message to *aClipView* with *newOrigin* as the argument. This mechanism is provided so that the NSClipView's superview can coordinate scrolling of multiple tiled NSClipViews.

See also: – **scrollToPoint:** (NSClipView)

scrollPoint:

– (void)**scrollPoint:**(NSPoint)*aPoint*

Scrolls the receiver's closest ancestor NSClipView so that *aPoint* in the receiver lies at the origin of the NSClipView's bounds rectangle.

See also: – **autoscroll:**, – **scrollToPoint:** (NSClipView), – **isDescendantOf:**

scrollRect:by:

– (void)**scrollRect:**(NSRect)*aRect* **by:**(NSSize)*offset*

Copies the visible portion of the receiver's rendered image within *aRect* and lays that portion down again at *offset* from *aRect*'s origin. This method is useful during scrolling or translation of the coordinate system to efficiently move as much of the receiver's rendered image as possible without requiring it to be redrawn, following these steps:

1. Invoke **scrollRect:by:** to copy the rendered image.
2. Move the view object's origin or scroll it within its superview.
3. Calculate the newly exposed rectangles and invoke either **displayRect:** or **setNeedsDisplayInRect:** to draw them.

You should rarely need to use this method, however. The **scrollPoint:**, **scrollRectToVisible:**, and **autoscroll:** methods automatically perform optimized scrolling.

See also: – **setBoundsOrigin:**, – **translateOriginToPoint:**

scrollRectToVisible:

– (BOOL)**scrollRectToVisible:**(NSRect)*aRect*

Scrolls the receiver's closest ancestor NSClipView the minimum distance needed so that *aRect* in the receiver becomes visible in the NSClipView. Returns YES if any scrolling is performed; otherwise returns NO.

See also: – **autoscroll:**, – **scrollToPoint:** (NSClipView), – **isDescendantOf:**

setAutoresizesSubviews:

– (void)**setAutoresizesSubviews:**(BOOL)*flag*

Determines whether the receiver automatically resizes its subviews when its frame size changes. If *flag* is YES, the receiver invokes **resizeSubviewsWithOldSize:** whenever its frame size changes; if *flag* is NO, it doesn't. View objects by default do autoresize their subviews.

See also: – **autoresizesSubviews**

setAutoresizingMask:

– (void)**setAutoresizingMask:**(unsigned int)*mask*

Determines how the receiver's **resizeWithOldSuperviewSize:** method changes its frame rectangle. *mask* can be specified by combining any of the following options using the C bitwise OR operator:

Option	Meaning
NSViewMinXMargin	The left margin between the receiver and its superview is flexible.
NSViewWidthSizable	The receiver's width is flexible.
NSViewMaxXMargin	The right margin between the receiver and its superview is flexible.
NSViewMinYMargin	The bottom margin between the receiver and its superview is flexible.
NSViewHeightSizable	The receiver's height is flexible.
NSViewMaxYMargin	The top margin between the receiver and its superview is flexible.

Where more than one option along an axis is set, **resizeWithOldSuperviewSize:** by default distributes the size difference as evenly as possible among the flexible portions. For example, if NSViewWidthSizable and NSViewMaxXMargin are set and the superview's width has increased by 10.0 units, the receiver's frame and right margin are each widened by 5.0 units.

See also: – **autoresizingMask**, – **resizeSubviewsWithOldSize:**, – **setAutoresizesSubviews:**

setBounds:

– (void)**setBounds:**(NSRect)*boundsRect*

Sets the receiver's bounds rectangle to *boundsRect*. The bounds rectangle determines the origin and scale of the receiver's coordinate system within its frame rectangle. This method neither redisplays the receiver nor marks it as needing display. You must do this yourself with **display** or **setNeedsDisplay:**.

This method posts an NSViewBoundsDidChangeNotification to the default notification center if the receiver is configured to do so.

See also: – **bounds**, – **setBoundsRotation:**, – **setBoundsOrigin:**, – **setBoundsSize:**, – **setFrame:**, – **setPostsBoundsChangedNotifications:**

setBoundsOrigin:

– (void)**setBoundsOrigin:**(NSPoint)*newOrigin*

Sets the origin of the receiver's bounds rectangle to *newOrigin*, effectively shifting its coordinate system so that *newOrigin* lies at the origin of the receiver's frame rectangle. This method neither redisplays the receiver nor marks it as needing display. You must do this yourself with **display** or **setNeedsDisplay:**.

This method posts an NSViewBoundsDidChangeNotification to the default notification center if the receiver is configured to do so.

See also: – **translateOriginToPoint:**, – **bounds**, – **setBoundsRotation:**, – **setBounds:**, – **setBoundsSize:**, – **setPostsBoundsChangedNotifications:**

setBoundsRotation:

– (void)**setBoundsRotation:**(float)*angle*

Sets the rotation of the receiver's bounds rectangle to *angle*. Positive values indicate counterclockwise rotation, negative clockwise. Rotation is performed around the coordinate system origin, (0.0, 0.0), which need not coincide with that of the frame rectangle or the bounds rectangle. This method neither redisplays the receiver nor marks it as needing display. You must do this yourself with **display** or **setNeedsDisplay:**.

This method posts an NSViewBoundsDidChangeNotification to the default notification center if the receiver is configured to do so.

Bounds rotation affects the orientation of the drawing within the view object's frame rectangle, but not the orientation of the frame rectangle itself. Also, for a rotated bounds rectangle to enclose all the visible areas of its view object—that is, to guarantee coverage over the frame rectangle—it must also contain some areas that aren't visible. This can cause unnecessary drawing to be requested, which may affect performance. It may be better in many cases to rotate the PostScript coordinate system in the **drawRect:** method rather than use this method.

See also: – **rotateByAngle:**, – **boundsRotation**, – **setFrameRotation:**, – **setPostsBoundsChangedNotifications:**

setBoundsSize:

– (void)**setBoundsSize:**(NSSize)*newSize*

Sets the size of the receiver's bounds rectangle to *newSize*, inversely scaling its coordinate system relative to its frame rectangle. For example, a view object with a frame size of (100.0, 100.0) and a bounds size of (200.0, 100.0) draws half as wide along the *x* axis. This method neither redisplays the receiver nor marks it as needing display. You must do this yourself with **display** or **setNeedsDisplay:**.

This method posts an NSViewFrameDidChangeNotification to the default notification center if the receiver is configured to do so.

See also: – **bounds**, – **setBoundsRotation:**, – **setBounds:**, – **setBoundsOrigin:**, – **setPostsBoundsChangedNotifications:**

setFrame:

– (void)**setFrame:**(NSRect)*frameRect*

Sets the receiver's frame rectangle to *frameRect*, thereby repositioning and resizing it within the coordinate system of its superview. This method neither redisplays the receiver nor marks it as needing display. You must do this yourself with **display** or **setNeedsDisplay:**.

This method posts an NSViewFrameDidChangeNotification to the default notification center if the receiver is configured to do so.

See also: – **frame**, – **setFrameRotation:**, – **setFrameOrigin:**, – **setFrameSize:**, – **setBounds:**, – **setPostsFrameChangedNotifications:**

setFrameOrigin:

– (void)**setFrameOrigin:**(NSPoint)*newOrigin*

Sets the origin of the receiver's frame rectangle to *newOrigin*, effectively repositioning it within its superview. This method neither redisplays the receiver nor marks it as needing display. You must do this yourself with **display** or **setNeedsDisplay:**.

This method posts an NSViewFrameDidChangeNotification to the default notification center if the receiver is configured to do so.

See also: – **frame**, – **setFrameSize:**, – **setFrame:**, – **setFrameRotation:**, – **setPostsFrameChangedNotifications:**

setFrameRotation:

– (void)**setFrameRotation:**(float)*angle*

Sets the rotation of the receiver's frame rectangle to *angle*, rotating it within its superview without affecting its coordinate system. Positive values indicate counterclockwise rotation, negative clockwise. Rotation is performed around the origin of the frame rectangle.

This method neither redisplays the receiver nor marks it as needing display. You must do this yourself with **display** or **setNeedsDisplay:**.

This method posts an NSViewFrameDidChangeNotification to the default notification center if the receiver is configured to do so.

See also: – **frameRotation**, – **setBoundsRotation:**

setFrameSize:

– (void)**setFrameSize:**(NSSize)*newSize*

Sets the size of the receiver's frame rectangle to *newSize*, resizing it within its superview without affecting its coordinate system. This method neither redisplays the receiver nor marks it as needing display. You must do this yourself with **display** or **setNeedsDisplay:**.

This method posts an NSViewFrameDidChangeNotification to the default notification center if the receiver is configured to do so.

See also: – **frame**, – **setFrameOrigin:**, – **setFrame:**, – **setFrameRotation:**, – **setPostsFrameChangedNotifications:**

setInterfaceStyle:

– (void)**setInterfaceStyle:(**NSInterfaceStyle**)***interfaceStyle*

Sets the interface style for the view and for its subviews to the style specified by *interfaceStyle*, such as NSMacintoshInterfaceStyle or NSWindows95InterfaceStyle. You should almost never need to invoke or override this method, but if you do override it, your version should always invoke **super**.

See also: – **interfaceStyle**

setNeedsDisplay:

– (void)**setNeedsDisplay:**(BOOL)*flag*

If *flag* is YES, marks the receiver's entire bounds as needing display; if *flag* is NO, marks it as not needing display. Whenever the data or state used for drawing a view object changes, the view should be sent a **setNeedsDisplay:** message. NSViews marked as needing display are automatically redisplayed on each pass through the application's event loop. (View objects that need to redisplay before the event loop comes around can of course immediately be sent the appropriate **display...** method.)

See also: – **setNeedsDisplayInRect:**, – **needsDisplay**

setNeedsDisplayInRect:

– (void)**setNeedsDisplayInRect:**(NSRect)*invalidRect*

Marks the region of the receiver within *invalidRect* as needing display, increasing the receiver's existing invalid region to include it. A later **displayIfNeeded...** method will then perform drawing only within the invalid region. NSViews marked as needing display are automatically redisplayed on each pass through the application's event loop. (View objects that need to redisplay before the event loop comes around can of course immediately be sent the appropriate **display...** method.)

See also: – **setNeedsDisplay:**, – **needsDisplay**

setNextKeyView:

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

Inserts *aView* after the receiver in the key view loop of the receiver's NSWindow.

See also: – **nextKeyView**, – **nextValidKeyView**, – **previousKeyView**, – **previousValidKeyView**

setPostsBoundsChangedNotifications:

– (void)**setPostsBoundsChangedNotifications:**(BOOL)*flag*

Controls whether the receiver informs observers when its bounds rectangle changes. If *flag* is YES, the receiver will post notifications to the default notification center whenever its bounds rectangle changes; if *flag* is NO it won't. The following methods can result in notification posting:

setBounds:
setBoundsOrigin:
setBoundsSize:
setBoundsRotation:
translateOriginToPoint:
scaleUnitSquareToSize:
rotateByAngle:

See also: – **postsBoundsChangedNotifications**

setPostsFrameChangedNotifications:

– (void)**setPostsFrameChangedNotifications:**(BOOL)*flag*

Controls whether the receiver informs observers when its frame rectangle changes. If *flag* is YES, the receiver will post notifications to the default notification center whenever its frame rectangle changes; if *flag* is NO it won't. The following methods can result in notification posting:

setFrame:
setFrameOrigin:
setFrameSize:
setFrameRotation:

See also: – **postsFrameChangedNotifications**

setToolTip:

– (void)**setToolTip:**(NSString *)*string*

Sets the tooltip text for the view according to the passed *string*. If *string* is **nil**, cancels tooltip display for the view.

See also: – **toolTip**

setUpGState

– (void)**setUpGState**

Overridden by subclasses to (re)initialize the receiver's graphics state object. This method is automatically invoked when the graphics state object created using **allocateGState** needs to be initialized. NSView's implementation does nothing. Your subclass can override it to set the current font, line width, or any other PostScript graphics state parameter except coordinate transformations and the clipping path—these are established by the frame and bounds rectangles, and by methods such as **scaleUnitSquareToSize:** and **translateOriginToPoint:**. Note that **drawSelf:** can further transform the coordinate system and clipping path for whatever temporary effects it needs.

See also: – **allocateGState**, – **renewGState**

shouldDelayWindowOrderingForEvent:

– (BOOL)**shouldDelayWindowOrderingForEvent:**(NSEvent *)*theEvent*

Overridden by subclasses to allow the user to drag images from the receiver without its window moving forward and possibly obscuring the destination, and without activating the application. If this method returns YES, the normal window ordering and activation mechanism is delayed (*not* necessarily prevented) until the next mouse-up event. If it returns NO then normal ordering and activation occurs. Never invoke this method directly; it's invoked automatically for each mouse-down event directed at the NSView.

An NSView subclass that allows dragging should implement this method to return YES if *theEvent*, an initial mouse-down event, is potentially the beginning of a dragging session or of some other context where window ordering isn't appropriate. This method is invoked before a **mouseDown:** message for *theEvent* is sent. NSView's implementation returns NO.

If, after delaying window ordering, the receiver actually initiates a dragging session or similar operation, it should also send a **preventWindowOrdering** message to NSApp, which completely prevents the window from ordering forward and the activation from becoming active. **preventWindowOrdering** is sent automatically by NSView's **dragImage:...** and **dragFile:...** methods.

shouldDrawColor

– (BOOL)**shouldDrawColor**

Returns NO if the receiver is being drawn in an NSWindow (as opposed, for example, to being printed) and the NSWindow can't store color; otherwise returns YES. An NSView can base its drawing behavior on the return value of this method to improve its appearance in grayscale windows.

See also: – **drawRect:**, – **canStoreColor** (NSWindow)

sortSubviewsUsingFunction:context:

– (void)**sortSubviewsUsingFunction:**(int (*)(id, id, void *))*compare* **context:**(void *)*context*

Orders the receivers immediate subviews using the comparator function *compare*, which takes as arguments two subviews to be ordered and the *context* supplied, which may be arbitrary data used to help in the decision. *compare* should return NSOrderedAscending if the first subview should be ordered lower, NSOrderedDescending if the second subview should be ordered lower, and NSOrderedSame if their ordering isn't important.

See also: – **sortedArrayUsingFunction:context:** (NSArray class cluster of the Foundation Kit)

subviews

– (NSArray *)**subviews**

Return the receiver's immediate subviews.

See also: – **superview**, – **addSubview:**, – **addSubview:positioned:relativeTo:**, – **removeFromSuperview**

superview

– (NSView *)**superview**

Returns the receiver's superview, or **nil** if it has none. When applying this method iteratively or recursively, be sure to compare the returned NSView to the content view of the NSWindow to avoid proceeding out of the view hierarchy.

See also: – **window**, – **subviews**, – **removeFromSuperview**

tag

– (int)**tag**

Returns the receiver's tag, an integer that you can use to identify view objects in your application. NSView's implementation returns –1. Subclasses can override this method to provide individual tags, possibly adding storage and a **setTag:** method (which NSView doesn't define).

See also: – **viewWithTag:**

toolTip

– (NSString *)**toolTip**

Returns the text for the view's tool tip. Returns **nil** if the view doesn't currently display tooltip text.

See also: – **setToolTip:**

translateOriginToPoint:

– (void)**translateOriginToPoint:**(NSPoint)*newOrigin*

Translates the receiver's coordinate system so that its origin moves to *newOrigin*. In the process, the origin of the receiver's bounds rectangle is shifted by (*–newOrigin.x*, *–newOrigin.y*). This method neither redisplays the receiver nor marks it as needing display. You must do this yourself with **display** or **setNeedsDisplay:**.

Note the difference between this method and setting the bounds origin. Translation effectively moves the image inside the bounds rectangle, while setting the bounds origin effectively moves the rectangle over the image. The two are in a sense inverse, although translation is cumulative and setting the bounds origin is absolute.

This method posts an NSViewBoundsDidChangeNotification to the default notification center if the receiver is configured to do so.

See also: – **setBoundsOrigin:**, – **setBounds:**, – **setPostsBoundsChangedNotifications:**

unlockFocus

– (void)**unlockFocus**

Balances an earlier **lockFocus** message, restoring the focus to the previously focused view is necessary.

See also: – **allocateGState**

unregisterDraggedTypes

– (void)**unregisterDraggedTypes**

Unregisters the receiver as a possible destination in a dragging session.

See also: – **registerForDraggedTypes:**

viewWillMoveToSuperview:

– (void)**viewWillMoveToSuperview:**(NSView *)*newSuperview*

Informs the receiver that it's being added as a subview of *newSuperview*. Subclasses can override this method to perform whatever actions are necessary.

See also: – **viewWillMoveToWindow:**

viewWillMoveToWindow:

– (void)**viewWillMoveToWindow:**(NSWindow *)*newWindow*

Informs the receiver that it's being added to the view hierarchy of *newWindow*. Subclasses can override this method to perform whatever actions are necessary.

See also: – **viewWillMoveToSuperview:**

viewWithTag:

– (id)**viewWithTag:**(int)*aTag*

Returns the receiver's nearest descendant (including itself) whose tag is *aTag*, or **nil** if no subview has that tag.

See also: – **tag**

visibleRect

– (NSRect)**visibleRect**

Returns the portion of the receiver not clipped by its superviews. Visibility is therefore defined quite simply, and doesn't account for whether other NSViews (or windows) overlap the receiver or whether the receiver has a window at all.

Note: During a printing operation the visible rectangle is further clipped to the page being imaged.

See also: – **isVisible** (NSWindow), – **documentVisibleRect** (NSScrollView),
– **documentVisibleRect** (NSClipView)

widthAdjustLimit

– (float)**widthAdjustLimit**

Returns the fraction (between 0.0 and 1.0) of the page that can be pushed onto the next page during automatic pagination to prevent items such as small images or text columns from being divided across

pages. This fraction is used to calculate the right edge limit for a **adjustPageWidthNew:left:right:limit:** message.

See also: – **heightAdjustLimit**

window

– (NSWindow *)**window**

Returns the receiver's window object, or **nil** if it has none.

See also: – **superview**

writeEPSInsideRect:toPasteboard:

– (void)**writeEPSInsideRect:**(NSRect)*aRect* **toPasteboard:**(NSPasteboard *)*pboard*

Writes EPS data that draws the region of the receiver within *aRect* onto *pboard*.

See also: – **dataWithEPSInsideRect:**

Notifications

NSViewBoundsDidChangeNotification

Posted whenever the NSView's bounds rectangle changes independently of the frame rectangle, if the NSView is configured using **setPostsBoundsChangedNotifications:** to post such notifications.

This notification contains a notification object but no userInfo dictionary. The notification object is the NSView whose bounds rectangle has changed.

The following methods can result in notification posting:

setBounds:
setBoundsOrigin:
setBoundsSize:
setBoundsRotation:
translateOriginToPoint:
scaleUnitSquareToSize:
rotateByAngle:

Note that the bounds rectangle resizes automatically to track the frame rectangle. Because the primary change is that of the frame rectangle, however, **setFrame:** and **setFrameSize:** don't result in a bounds-changed notification.

NSViewFocusDidChangeNotification

Posted whenever the NSView loses the PostScript focus other than by an **unlockFocus** message (for example, when its frame or bounds rectangle is changed).

This notification contains a notification object but no userInfo dictionary. The notification object is the NSView that has lost focus.

See also: + **focusView**

NSViewFrameDidChangeNotification

Posted whenever the NSView's frame rectangle changes, if the NSView is configured using **setPostsFrameChangedNotifications:** to post such notifications.

This notification contains a notification object but no userInfo dictionary. The notification object is the NSView whose frame rectangle has changed.

The following methods can result in notification posting:

setFrame:
setFrameOrigin:
setFrameSize:
setFrameRotation:

NSWindow

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

Class at a Glance

Purpose

An NSWindow manages an on-screen window, coordinating the display and event handling for its NSViews. Interface Builder allows you to create and set up NSWindows, but there are many things you may wish to do programmatically as well.

Principal Attributes

- Manages a view hierarchy
- Uses a delegate
- Distributes events to view objects
- Provides a field editor to view objects

Creation

Interface Builder

– initWithContentRect:styleMask:backing:defer: Designated initializer.

Commonly Used Methods

– makeKeyAndOrderFront:	Moves the NSWindow to the front and makes it the key window.
– makeFirstResponder:	Sets the first responder in the NSWindow.
– fieldEditor:forObject:	Returns the shared text object for the NSWindow.
– setContentView:	Sets the root-level NSView in the NSWindow.
– representedFilename	Returns the filename whose contents the NSWindow presents.
– setDocumentEdited:	Sets whether the NSWindow's represented file needs to be saved.
– setTitle:	Sets the title of the NSWindow.
– setTitleWithRepresentedFilename:	Sets the title of the NSWindow in a readable format for filenames.

Class Description

The NSWindow class defines objects that manage and coordinate the windows that an application displays on the screen. A single NSWindow object corresponds to at most one on-screen window. The two principal functions of NSWindow are to provide an area in which NSViews can be placed and to accept and distribute, to the appropriate NSViews, events that the user instigates through actions with the mouse and keyboard. Note that the term *window* sometimes refers to the Application Kit object and sometimes to the Window Server's PostScript window device; which meaning is intended is made clear in context. The Application Kit also defines an abstract subclass of NSWindow—NSPanel—that adds behavior more appropriate for auxiliary windows.

You typically set windows up using Interface Builder, which allows you to position them, set up many of their visual and behavioral attributes, and lay out views in them. The programmatic work you do with windows more often involves bringing them on and off the screen; changing dynamic attributes such as the window's title; running modal windows to restrict user input; and assigning a delegate that can monitor certain of the window's actions, such as closing, zooming, and resizing.

Window Anatomy

An NSWindow is defined by a *frame rectangle* that encloses the entire window, including its title bar, border, and other peripheral elements (such as the resize bar on OPENSTEP for Mach), and by a *content rectangle* that encloses just its content area. Both rectangles are specified in the screen coordinate system and restricted to integer values. The frame rectangle establishes the NSWindow's *base coordinate system*. This coordinate system is always aligned with and measured in the same increments as the screen coordinate system (in other words, the base coordinate system can't be rotated or scaled). The origin of the base coordinate system is the bottom-left corner of the NSWindow's frame rectangle.

You create an NSWindow programmatically through one of the **initWithContentRect:...** methods by specifying, among other attributes, the size and location of its content rectangle. The frame rectangle is derived from the dimensions of the content rectangle. Various sections below describe other attributes you can specify at initialization and afterward.

When it's created, an NSWindow automatically creates two NSViews: An opaque *frame view* that fills the frame rectangle and draws the border, title bar, other peripheral elements, and background, and a transparent *content view* that fills the content rectangle. The frame view and its peripheral elements are private objects that your application can't access directly. The content view is the "highest" accessible NSView in the NSWindow; you can replace the default content view with an NSView of your own creation using the **setContentView:** method. The NSWindow determines the placement of the content view; you can't position it using NSView's **setFrame...** methods, but must use NSWindow's placement methods, described in "Windows on the Screen" below.

You add other NSViews to the NSWindow as subviews of the content view or as subviews of any of the content view's subviews, and so on, through NSView's **addSubview:** method. This tree of NSViews is called the NSWindow's *view hierarchy*. When an NSWindow is told to display itself, it does so by sending **display...** messages to the top-level NSView in its view hierarchy. Because displaying is carried out in a

determined order, the content view (which is drawn first) may be wholly or partially obscured by its subviews, and these subviews may be obscured by their subviews (and so on).

Window Styles

The peripheral elements that an NSWindow displays define its style. Though you can't access and manipulate them directly, you can determine at initialization whether an NSWindow has them by providing a style mask to the **initWithContentRect:styleMask:backing:defer:** method. There are four possible style elements, specifiable by combining their mask values using the C bitwise OR operator:

Element	Mask Value
A title bar	NSTitledWindowMask
A close button	NSClosableWindowMask
A miniaturize button	NSMiniaturizableWindowMask
A resize bar, border, or box	NSResizableWindowMask

You can also specify NSBorderlessWindowMask, in which case none of these style elements is used.

Windows on the Screen

NSWindows can be placed on the screen in three dimensions. Besides horizontal and vertical placement, NSWindows are ordered back-to-front in several distinct *levels*, which group windows of similar type and purpose so that the more "important" ones appear in front of those less so. (For more information, see the description for **setLevel:**.) Placing an NSWindow on the screen is accomplished with the **setFrame: display:** method and its variants **setFrameOrigin:** and **setFrameTopLeftPoint:**. Ordering takes place in two ways. The **setLevel:** method puts an NSWindow into a group, such as that for standard windows, floating windows (for example, palettes and some inspector panels), menus, and so on. The **orderWindow: relativeTo:** method orders an NSWindow within its level in front of or in back of another. Convenience methods for ordering include **makeKeyAndOrderFront:**, **orderFront:** and **orderBack:**, as well as **orderOut:**, which removes an NSWindow from the screen. The **isVisible** method tells whether an NSWindow is on or off the screen. You can also set a window to be removed from the screen automatically when its application isn't active using **setHidesOnDeactivate:**.

NSWindow offers several means of constraining and adjusting window placement:

> **setMinSize:** and **setMaxSize:** limit the user's ability to resize the NSWindow (you can still set it to any size programmatically). Similarly, **setAspectRatio:** keeps a window's width and height at the same proportions as the user resizes it, and **setResizeIncrements:** makes the window resize in discrete amounts larger than a single pixel. (Aspect ratio and resize increments are mutually

exclusive attributes. See the **setAspectRatio:** and **setResizeIncrements:** method descriptions for more information.)

The **constrainFrameRect:toScreen:** method adjusts a proposed frame rectangle so that it lies on the screen in such a way that the user can move and resize a window. Note that any NSWindow with a title bar automatically constrains itself to the screen. The **cascadeTopLeftFromPoint:** method shifts the top left point by an amount that allows one NSWindow to be placed relative to another so that both their title bars are visible.

The **zoom:** method toggles the size and location of a window between its standard state, as determined by the application, and its user state—a new size and location the user may have set by moving or resizing the window.

Finally, the **center** method places an NSWindow in the most prominent location on the screen, one suitable for important messages and attention panels.

Closely related to window ordering is the idea of opening or closing an NSWindow. Normally, opening is accomplished simply by ordering the NSWindow in front of or in back of another that's on screen. Closing a window involves explicit use of either the **close** method, which simply removes the NSWindow from the screen, or **performClose:**, which highlights the close button as though the user clicked it. Closing an NSWindow involves at least removing it from the screen but adds the possibility of disposing of it altogether. The **setReleasedWhenClosed:** method sets whether an NSWindow releases itself when it receives a close message. An NSWindow's delegate is also notified when it's about to close, as described in the "Notifications and the NSWindow's Delegate" section.

Miniaturizable windows can be removed from the screen and replaced by a smaller counterpart, whether a freestanding miniwindow or, on Microsoft Windows, a button in the task bar. The **miniaturize:** and **deminiaturize:** methods reduce and reconstitute an NSWindow, and **performMiniaturize:** simulates the user clicking the NSWindow's miniaturize button. You can also set the image and title displayed in a freestanding miniwindow by sending **setMiniwindowImage:** and **setMiniwindowTitle:** messages to the NSWindow object.

An NSWindow can store its placement in the user defaults system, so that it appears in the same location the next time the user starts the application. The **saveFrameUsingName:** method stores the frame rectangle, and **setFrameUsingName:** sets it from the value in user defaults. You can also use the **setFrameAutosaveName:** method to have an NSWindow save the frame rectangle any time it changes. To expunge a frame rectangle from the defaults system, use the class method **removeFrameUsingName:**.

Titles and Represented Files

A titled NSWindow can display an arbitrary title or one derived from a filename. The **setTitle:** method puts an arbitrary string on the title bar. The **setTitleWithRepresentedFilename:** method formats a filename in the title bar in a readable format (which varies with the platform) and associates the NSWindow with that file. You can set the associated file without changing the title using **setRepresentedFilename:**. You can use the association between the NSWindow and the file in any way you see fit. One convenience offered by NSWindow is marking the file as being edited, so that you can prompt the user to save it on closing the

window. The method for marking this is **setDocumentEdited:**. When the window closes, its delegate can check it with **isDocumentEdited** to see whether the document needs to be saved.

Most OPENSTEP applications include a submenu that displays the titles of windows, called the *Windows menu*. This submenu automatically lists windows that have a title bar and are resizable and that can become the main window (as described under "Event Handling"). When you change an NSWindow's title, this change is also automatically reflected in the Windows menu. You can exclude a window that would otherwise be listed by sending it a **setExcludedFromWindowsMenu:** message.

Window Device Attributes

Nearly every NSWindow has a corresponding PostScript window device in the Window Server. The window device holds the NSWindow's drawn image, and has two attributes determined by the Window Server and five attributes that the NSWindow controls. The Window Server assigns the window device a unique identifier (within an application). This is the *window number*, and it's returned by the **windowNumber** method. Each window also has a PostScript graphics state that most NSViews share for drawing (NSViews can create their own as well). The **gstate** method returns its identifier. The five attributes under direct NSWindow control are:

- Where the drawn image is stored, determined by the window's *backing store type*
- When the window device is created
- Whether the window device persists when the window is off screen
- How much memory is used for each pixel (also called the *depth limit*)
- Whether the depth limit changes with the screen capacity

A window device's backing store type is set when the NSWindow is initialized and can be one of three types. A *buffered* window device renders all drawing into a display buffer and then flushes it to the screen. Always drawing to the buffer produces very smooth display, but can require significant amounts of memory. Buffered windows are best for displaying material that must be redrawn often, such as text. A *retained* window device also uses a buffer, but draws directly to the screen where possible and to the buffer for any portions that are obscured. A *nonretained* window device has no buffer at all, and must redraw portions as they're exposed. Further, this redrawing is suspended when the NSWindow's display mechanism is preempted. For example, if the user drags a window across a nonretained window, the nonretained window is "erased" and isn't redrawn until the user releases the mouse. Both retained and nonretained windows are also subject to a flashing effect as individual drawing operations are performed, but their results do get to the screen more quickly than those of buffered windows. You can change the backing store type between buffered and retained after initialization using the **setBackingType:** method.

The last argument to **initWithContentRect:styleMask:backing:defer:** specifies whether the NSWindow creates its window device immediately or only when it's moved on screen. Deferring creation of the window device can offer some performance gain for windows that aren't displayed immediately because it reduces the amount of work that needs to be performed up front. Deferring creation of the window device is particularly useful when creation of the NSWindow itself can't be deferred or when an NSWindow is needed for purposes other than displaying content. Submenus with key equivalents, for example, must exist for the key equivalents to work, but may never actually be displayed.

Memory can also be saved by destroying the window device when the window is removed from the screen. The **setOneShot:** method controls this behavior. One-shot window devices exist only when their NSWindows are on screen.

Like the display hardware, a window device's buffer has a depth, or a limit to the memory allotted each pixel. Buffered and retained windows start out with a default window depth of 2 bits per pixel, and this depth grows to the window device's limit as the NSWindow draws richer images (more shades of gray, more colors). A window device's depth is set using the **setDepthLimit:** method, which takes as an argument a window depth limit created using the **NSBestDepth** function.

If an NSWindow draws color into its buffer and there's a color screen available, the Window Server automatically promotes the window's depth (up to its limit). This happens whether or not the window is actually on a color screen; similarly, if the user drags a window that displays color from a color screen to a monochrome screen, it remains at its richer depth. In both cases, the window's depth is greater than the screen can properly display. Keeping a window's depth at its richest preserves the displayed image, but it may produce undesired results such as dithering on a more limited screen and does causes slight performance reduction when the window buffer is deeper than the screen requires. You can set an NSWindow to keep its depth at the limit of the screen it's on with the **setDynamicDepthLimit:** method. When it's moved to a new screen, a window with a dynamic depth limit is redrawn into the newly adjusted buffer. Making a window's depth limit dynamic overrides the limit set using **setDepthLimit:**, and removing the dynamic limit reverts the static limit to the default.

Window Display and Updating

Display of an NSWindow begins with the drawing performed by its view objects, which accumulates in the window's display buffer or appears immediately on the screen. NSWindows, like NSViews, can be displayed unconditionally or merely marked as needing display, using the **display** and **setViewsNeedDisplay:** methods, respectively. A **displayIfNeeded** message causes the NSWindow's views to display only if they've been marked as needing display. Normally, any time an NSView is marked as needing display, the NSWindow makes note of this fact and automatically displays itself shortly after. This automatic display is typically performed on each pass through the event loop, but can be turned off using the **setAutodisplay:** method. If you turn off autodisplay for an NSWindow, you're then responsible for displaying it whenever necessary.

A related mechanism is that of updating. On each pass through the event loop, the application object invokes its **updateWindows** method, which sends an **update** message to each NSWindow. Subclasses of NSWindow can override this method to examine the state of the application and change their own state or appearance accordingly—enabling or disabling menus, buttons, and other controls based on the object that's selected, for example.

In addition to displaying itself on the screen, an NSWindow can print itself in its entirety, just as an NSView can. The **print:** method runs the application's Print panel and causes the NSWindow's frame view to print itself. The **fax:** and **dataWithEPSInsideRect:** methods behave similarly. See the NSView class specification for more information on printing.

Event Handling

As described in the NSResponder class specification, most events coming into an application make their way to an NSWindow in a **sendEvent:** message. A key event is directed at the key window, while a mouse event is directed at whatever window lies under the cursor. If an event affects the window directly—resizing or moving it, for example—it performs the appropriate operation itself and sends messages to its delegate informing it of its intentions, thus allowing your application to intercede. The window sends other events up its responder chain from the appropriate starting point: the first responder for a key event, the view under the cursor for a mouse event. These events are then typically handled by some view object in the window. See the NSView and NSEvent class specifications for more information on how to intercept and handle events.

The following sections describe aspects of events not directly related to handling individual events. These include changing the key and main windows, changing the first responder by keyboard rather than mouse actions, sharing a single text object for lightweight editing tasks, and running a modal event loop around an entire window rather than a single view object.

Changing the Key and Main Windows

Windows already on screen automatically change their status as the key or main window based on the user's actions with the mouse and on how the clicked view handles the mouse event. You can also set the key and main windows programmatically by sending the relevant window object a **makeKeyWindow** or **makeMainWindow** message. Setting the key and main windows programmatically is particularly useful when creating a new window. Because making a window key is often combined with ordering the window to the front of the screen, NSWindow defines a convenience method, **makeKeyAndOrderFront:**, that performs both operations.

Not all windows are suitable for acting as the key or main window. For example, a window that merely displays information, and contains no objects that need to respond to events or action messages, can completely forgo ever becoming the key window. Similarly, a window that acts as a floating palette of items that are only dragged out by mouse actions never needs to be the key window. Such a window can be defined as a subclass of NSWindow that overrides the methods **canBecomeKeyWindow** and **canBecomeMainWindow** to return NO instead of the default of YES. Defining a window in this way prevents it from ever becoming the key or main window. Though NSWindow defines these methods, typically only subclasses of NSPanel refuse to accept key or main window status.

Keyboard Interface Control

A window's first responder is often a view object selected by the user clicking it. For text fields and other view objects (mainly subclasses of NSControl), the user can select the first responder with the keyboard using the Tab and Shift keys. NSView defines the methods for setting up and examining the loop of objects that the user can select in this manner. A view that's the first responder is called the *key view*, and the views that can become the key view in a window are linked together in the window's *key view loop*. You normally set up the key view loop using Interface Builder, establishing connections between the **nextKeyView** outlets of views in the window and setting the window's **initialFirstResponder** outlet to the view that you

want selected when the window is first placed on screen. If you do not set this outlet, the window sets a key loop (not necessarily the same as the one you may have specified!) and picks a default initial first responder for you.

In addition to the key view loop, a window can have a default button cell, which uses the Return (or Enter) key as its key equivalent. The **setDefaultButtonCell:** method establishes this button cell; you can also set it in Interface Builder by setting a button cell's key equivalent to '\r'. The default button cell draws itself as a focal element for keyboard interface control unless another button cell is focused on. In this case, it temporarily draws itself as normal and disables its key equivalent. Another default key established by NSWindow is the Escape key, which immediately aborts a modal loop (described below under "Modal Windows").

See the NSResponder class specification for more information on keyboard interface control.

The Field Editor

Each NSWindow has a text object that is shared for light editing tasks. This object, the window's *field editor*, is inserted in the view hierarchy when an object needs to edit some text and removed when the object is finished. The field editor is used by NSTextFields and other controls, for example, to edit the text that they display. The **fieldEditor:forObject:** method returns a window's field editor, after asking the delegate for a substitute using **windowWillReturnFieldEditor:toObject:**. You can override the NSWindow method in subclasses or provide a delegate to substitute a class of text object different from the default of NSTextView, thereby customizing text editing in your application.

Modal Windows

You can make a whole window or panel run in modal fashion, using the application's normal event loop machinery but restricting input to the modal window or panel. Modal operation is useful for windows and panels that require the user's attention before an action can proceed. Examples include error messages and warnings, as well as operations that require input, such as printing or saving a document.

There are two mechanisms for operating a modal window or panel. The first, and simpler, is to invoke NSApplication's **runModalForWindow:** method, which monopolizes events for the specified window until one of the NSApplication methods **stopModal**, **abortModal**, or **stopModalWithCode:** is invoked, typically by a button's action method. The **stopModal** method ends the modal status of the window or panel from within the event loop. It doesn't work if invoked from a method invoked by a timer or by a distributed object because those mechanisms operate outside of the event loop. To terminate the modal loop in these situations, you can use **abortModal**. The **stopModal** method is typically invoked when the user clicks the OK button (or equivalent), **abortModal** when the user clicks the Cancel button (or presses the Escape key). These two methods are equivalent to **stopModalWithCode:** with the appropriate argument. See the method descriptions in the NSApplication class specification for more information.

The second mechanism for operating a modal window or panel, called a *modal session*, allows the application to perform a long operation while it still sends events to the window or panel. Modal sessions are particularly useful for panels that allow the user to cancel or modify an operation. To begin a modal

session, invoke NSApplication's **beginModalSessionForWindow:** method, which sets the window up for the session and returns an identifier used for other session-controlling methods. At this point, the application can run in a loop that performs the operation, on each pass sending **runModalSession:** to the application object so that pending events can be dispatched to the modal window. This method returns a code indicating whether the operation should continue, stop, or abort, which is typically established by the methods described above for **runModalForWindow:**. After the loop concludes, you can remove the window from the screen and invoke NSApplication's **endModalSession:** method to restore the normal event loop. The method description for **runModalForWindow** in the NSApplication class specification includes sample code illustrating a modal session.

Note: You can write a modal event loop for a view object so that the object has access to all events pertaining to a particular task, such as tracking the mouse in the view. For an example, see "Mouse Events" in the class specification for NSView.

The normal behavior of a modal window or session is to exclude all other windows and panels from receiving events. For windows and panels that serve as general auxiliary controls, such as menus and the Font panel, this behavior is overly restrictive. The user must be able to use menu key equivalents (such as those for Cut and for Paste) and change the font of text in the modal window, and this requires that nonmodal panels be able to receive events. To support this behavior, an NSWindow subclass overrides the **worksWhenModal** method to return YES. This allows the window to receive mouse and keyboard events even when a modal window is present. If a subclass needs to work when a modal window is present, it should usually be a subclass of NSPanel, not of NSWindow.

Modal windows and modal sessions provide different levels of control to the application and the user. Modal windows restrict all action to the window itself and any methods invoked from the window. Modal sessions allow the application to continue an operation while accepting input only through the modal session window. Beyond this, you can use distributed objects to perform background operations in a separate thread, while allowing the user to perform other actions with any part of the application. The background thread can communicate with the main thread, allowing the application to display the status of the operation in a nonmodal panel, perhaps including controls to stop or affect the operation as it occurs. Note that because the Application Kit isn't thread-safe, the background thread should communicate with a designated object in the main thread that in turn interacts with the Application Kit.

Notifications and the NSWindow's Delegate

NSWindow offers observers a rich set of notifications, which it broadcasts on such occurrences as gaining or losing key or main window status, miniaturizing, moving or resizing, becoming exposed, and closing. Each notification is matched to a delegate method, so an NSWindow's delegate is automatically registered for all notifications that it has methods for. NSWindow also offers its delegate a few other methods, such as **windowShouldClose:**, which requests approval to close, **windowWillResize:toSize:**, which allows the delegate to constrain the NSWindow's size, **windowWillUseStandardFrame:defaultFrame:**, which allows the delegate to set the window frame for zooming, and **windowWillReturnFieldEditor:toObject**, which gives the delegate a chance to modify the field editor or substitute a different editor. See the individual notification and delegate method descriptions at the end of this specification for more information.

Other Features

NSWindow defines a number of methods to assist its view objects in certain operations that may extend in scope beyond a single view or even outside the window containing them. One of these operations is image dragging. Although most dragging operations are initiated by and occur between view objects, NSWindow also defines an image-dragging method, **dragImage:at:offset:event:pasteboard:source:slideBack:**. An NSWindow can also serve as the destination for dragging operations, registering the types it accepts with **registerForDraggedTypes:** and **unregisterDraggedTypes**.

NSView provides methods for adding, removing, discarding, and resetting cursor rectangles—areas where the cursor image changes when the mouse enters them. NSWindow overrides some of these methods and provides additional methods for working with cursor rectangles. For example, an NSWindow can:

- disable and reenable all of its cursor rectangles with **disableCursorRects** and **enableCursorRects**
- determine if its cursor rects are enabled with **areCursorRectsEnabled**
- reset the cursor rectangles for a particular NSView with **invalidateCursorRectsForView:**
- reset all its cursor rectangles with **resetCursorRects**

Finally, to support transitory drawing by NSViews, NSWindow declares methods that temporarily cache a portion of its raster image so that it can be restored later. This feature is useful for situations where highly dynamic drawing must be done over the otherwise static image of the window. For example, in a drawing program where the user drags lines and other shapes directly onto a canvas, it’s more efficient to restore the window’s cached image and draw anew over that than to have all of the view objects send PostScript instructions to the Window Server. For more information, see the method descriptions for **cacheImageInRect:**, **restoreCachedImage**, and **discardCachedImage**.

Method Types

Creating instances

– initWithContentRect:styleMask:backing:defer:
– initWithContentRect:styleMask:backing:defer:screen:

Calculating layout

+ contentRectForFrameRect:styleMask:
+ frameRectForContentRect:styleMask:
+ minFrameWidthWithTitle:styleMask:

Converting coordinates

– convertBaseToScreen:
– convertScreenToBase:

Moving and resizing

– setFrame:display:
– frame
– setFrameOrigin:
– setFrameTopLeftPoint:
– setContentSize:
– cascadeTopLeftFromPoint:
– center
– resizeFlags
– performZoom:
– zoom:

Constraining window size

– maxSize
– minSize
– setMaxSize:
– setMinSize:
– setAspectRatio:
– aspectRatio
– setResizeIncrements:
– resizeIncrements
– constrainFrameRect:toScreen:

Saving the frame to user defaults

+ removeFrameUsingName:
– saveFrameUsingName:
– setFrameUsingName:
– setFrameAutosaveName:
– frameAutosaveName
– setFrameFromString:
– stringWithSavedFrame

Ordering windows

– orderBack:
– orderFront:
– orderFrontRegardless
– orderOut:
– orderWindow:relativeTo:
– setLevel:
– level
– isVisible

Making key and main windows

– becomeKeyWindow
– canBecomeKeyWindow
– isKeyWindow
– makeKeyAndOrderFront:
– makeKeyWindow
– resignKeyWindow
– becomeMainWindow
– canBecomeMainWindow
– isMainWindow
– makeMainWindow
– resignMainWindow

Working with the default button

– defaultButtonCell
– setDefaultButtonCell:
– disableKeyEquivalentForDefaultButtonCell
– enableKeyEquivalentForDefaultButtonCell

Display and drawing

– display
– displayIfNeeded
– setViewsNeedDisplay:
– viewsNeedDisplay
– useOptimizedDrawing:
– setAutodisplay:
– isAutodisplay
– update

Setting the interface style

– setInterfaceStyle:
– interfaceStyle

Flushing graphics

– flushWindow
– flushWindowIfNeeded
– enableFlushWindow
– disableFlushWindow
– isFlushWindowDisabled

Bracketing temporary drawing

– cacheImageInRect:
– restoreCachedImage
– discardCachedImage

Window Server information

– windowNumber
– gState
– deviceDescription
– setBackingType:
– backingType
– setOneShot:
– isOneShot
+ defaultDepthLimit
– setDepthLimit:
– depthLimit
– setDynamicDepthLimit:
– hasDynamicDepthLimit
– canStoreColor

Screen information

– deepestScreen
– screen

Working with the responder chain

– makeFirstResponder:
– firstResponder

Event handling

– currentEvent
– nextEventMatchingMask:
– nextEventMatchingMask:untilDate:inMode:dequeue:
– discardEventsMatchingMask:beforeEvent:
– postEvent:atStart:
– sendEvent:
– tryToPerform:with:
– keyDown:
– mouseLocationOutsideOfEventStream
– setAcceptsMouseMovedEvents:
– acceptsMouseMovedEvents

Working with the field editor

– fieldEditor:forObject:
– endEditingFor:

Keyboard interface control

– setInitialFirstResponder:
– initialFirstResponder
– selectKeyViewFollowingView:
– selectKeyViewPrecedingView:
– selectNextKeyView:
– selectPreviousKeyView:
– keyViewSelectionDirection

Setting the title and filename

– setTitle:
– setTitleWithRepresentedFilename:
– title
– setRepresentedFilename:
– representedFilename

Marking a window edited

– setDocumentEdited:
– isDocumentEdited

Closing the window

– close
– performClose:
– setReleasedWhenClosed:
– isReleasedWhenClosed

Miniaturizing and miniwindows

– miniaturize:
– performMiniaturize:
– deminiaturize:
– isMiniaturized
– setMiniwindowImage:
– miniwindowImage
– setMiniwindowTitle:
– miniwindowTitle

Working with menus

+ menuChanged:

Working with the Windows menu

– setExcludedFromWindowsMenu:
– isExcludedFromWindowsMenu

Working with cursor rectangles

– areCursorRectsEnabled
– enableCursorRects
– disableCursorRects
– discardCursorRects
– invalidateCursorRectsForView:
– resetCursorRects

Dragging

– dragImage:at:offset:event:pasteboard:source:slideBack:
– registerForDraggedTypes:
– unregisterDraggedTypes

Controlling behavior

– setHidesOnDeactivate:
– hidesOnDeactivate
– worksWhenModal

Setting the content view

– setContentView:
– contentView

Setting the background color

– setBackgroundColor:
– backgroundColor

Getting the style mask

– styleMask

Working with Services

– validRequestorForSendType:returnType:

Printing and faxing

– print:
– dataWithEPSInsideRect:
– fax:

Getting the Microsoft Windows handle

– windowHandle

Setting the delegate

– setDelegate:
– delegate

Class Methods

contentRectForFrameRect:styleMask:

+ (NSRect)**contentRectForFrameRect:**(NSRect)*frameRect* **styleMask:**(unsigned int)*aStyle*

Returns the content rectangle used by an NSWindow with a frame rectangle of *frameRect* and a style mask of *aStyle*. Both *frameRect* and the returned content rectangle are expressed in screen coordinates. See the **initWithContentRect:styleMask:backing:defer:** method description for a list of style mask values.

See also: + **frameRectForContentRect:styleMask:**

defaultDepthLimit

+ (NSWindowDepth)**defaultDepthLimit**

Returns the default depth limit for instances of NSWindow. This is determined by the depth of the deepest screen level available to the window server.

The value returned can be examined with the Application Kit functions **NSPlanarFromDepth**, **NSColorSpaceFromDepth**, **NSBitsPerSampleFromDepth**, **NSBitsPerPixelFromDepth**.

See also: – **setDepthLimit:**, – **setDynamicDepthLimit:**, – **canStoreColor**

frameRectForContentRect:styleMask:

+ (NSRect)**frameRectForContentRect:**(NSRect)*contentRect* **styleMask:**(unsigned int)*aStyle*

Returns the frame rectangle used by an NSWindow with a content rectangle of *contentRect* and a style mask of *aStyle*. Both *contentRect* and the returned frame rectangle are expressed in screen coordinates. See the **initWithContentRect:styleMask:backing:defer:** method description for a list of style mask values.

See also: + **contentRectForFrameRect:styleMask:**

menuChanged:

+ (void)**menuChanged:**(NSMenu *)*aMenu*

On Microsoft Windows, locates all objects inheriting from NSWindow that use *aMenu* and causes them to update their state and redisplay the menu. With other operating systems, this method does nothing.

See also: – **menu** (NSResponder)

minFrameWidthWithTitle:styleMask:

+ (float)**minFrameWidthWithTitle:**(NSString *)*aTitle* **styleMask:**(unsigned int)*aStyle*

Returns the minimum width that an NSWindow's frame rectangle must have for it to display all of *aTitle*, given *aStyle* as its style mask. See the **initWithContentRect:styleMask:backing:defer:** method description for a list of acceptable style mask values.

removeFrameUsingName:

+ (void)**removeFrameUsingName:**(NSString *)*name*

Removes the frame data stored under *name* from the application's user defaults.

See also: – **setFrameUsingName:**, – **setFrameAutosaveName:**

Instance Methods

acceptsMouseMovedEvents

– (BOOL)**acceptsMouseMovedEvents**

Returns YES if the receiver accepts and distributes mouse-moved events, NO if it doesn't. NSWindows by default don't accept mouse-moved events.

See also: – **setAcceptsMouseMovedEvents:**

areCursorRectsEnabled

– (BOOL)**areCursorRectsEnabled**

Returns YES if the receiver's cursor rectangles are enabled, NO if they're not.

See also: – **enableCursorRects**, – **addCursorRect:cursor:** (NSView)

aspectRatio

– (NSSize)**aspectRatio**

Returns the receiver's size aspect ratio. The size of the receiver's frame rectangle is constrained to integral multiples of this ratio when the user resizes it. You can set an NSWindow's size to any ratio programmatically.

See also: – **resizeIncrements**, – **setAspectRatio:**, – **setFrame:display:**

backgroundColor

– (NSColor *)**backgroundColor**

Returns the color of the receiver's background.

See also: – **setBackgroundColor:**

backingType

– (NSBackingStoreType)**backingType**

Returns the receiver's backing store type as one of the following constants:

NSBackingStoreBuffered
NSBackingStoreRetained
NSBackingStoreNonretained

See also: – **setBackingType:**

becomeKeyWindow

– (void)**becomeKeyWindow**

Invoked automatically to inform the receiver that it has become the key window; never invoke this method directly. This method reestablishes the receiver's first responder, sends the **becomeKeyWindow** message to that object if it responds, and posts an NSWindowDidBecomeKeyNotification to the default notification center.

See also: – **makeKeyWindow**, – **makeKeyAndOrderFront:**, – **becomeMainWindow**

becomeMainWindow

– (void)**becomeMainWindow**

Invoked automatically to inform the receiver that it has become the main window; never invoke this method directly. This method posts an NSWindowDidBecomeMainNotification to the default notification center.

See also: – **makeMainWindow**, – **becomeKeyWindow**

cacheImageInRect:

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

Stores the receiver's raster image from *aRect*, which is expressed in the receiver's base coordinate system. This allows the receiver to perform temporary drawing, such as a band around the selection as the user drags

the mouse, and to quickly restore the previous image by invoking **restoreCachedImage** and **flushWindowIfNeeded**. The next time the window displays, it discards its cached image rectangles. You can also explicitly use **discardCachedImage** to free the memory occupied by cached image rectangles.

See also: – **display**

canBecomeKeyWindow

– (BOOL)**canBecomeKeyWindow**

Returns YES if the receiver is able to be the key window, NO if it can't. Attempts to make the receiver the key window are abandoned if this method returns NO. NSWindow's implementation returns YES if the receiver has a title bar or a resize bar (size border in Windows), NO otherwise.

See also: – **isKeyWindow**, – **makeKeyWindow**

canBecomeMainWindow

– (BOOL)**canBecomeMainWindow**

Returns YES if the receiver is able to be the main window, NO if it can't. Attempts to make the receiver the main window are abandoned if this method returns NO. NSWindow's implementation returns YES if the receiver is visible, is *not* an NSPanel, and has a title bar or a resize mechanism. Otherwise it returns NO.

See also: – **isMainWindow**, – **makeMainWindow**

canStoreColor

– (BOOL)**canStoreColor**

Returns YES if the receiver has a depth limit that allows it to store color values, NO if it doesn't.

See also: – **depthLimit**, – **shouldDrawColor** (NSView)

cascadeTopLeftFromPoint:

– (NSPoint)**cascadeTopLeftFromPoint:**(NSPoint)*topLeftPoint*

Returns a point shifted from *topLeftPoint* that can be used to place the receiver in a cascade relative to another NSWindow positioned at *topLeftPoint*, so that the title bars of both NSWindows are fully visible. Both points are expressed in screen coordinates.

See also: – **setFrameTopLeftPoint:**

center

– (void)**center**

Sets the receiver's location to the center of the screen: The receiver is placed exactly in the center horizontally and somewhat above center vertically. Such a placement carries a certain visual immediacy and importance. This method doesn't put the receiver on screen, however; use **makeKeyAndOrderFront:** to do that.

You typically use this method to place a NSWindow—most likely an attention panel—where the user can't miss it. This method is invoked automatically when an NSPanel is placed on the screen by NSApplication's **runModalForWindow:** method.

close

– (void)**close**

Removes the receiver from the screen. If the receiver is set to be released when it's closed, a **release** message is sent to the object after the current event is completed. For an NSWindow object, the default is to be released on closing, while for an NSPanel object, the default is not to be released. You can use the **setReleasedWhenClosed:** method to change the default behavior.

A window doesn't have to be visible to receive the **close** message. For example, when the application terminates, it sends the **close** message to all windows in its window list, even those that are not currently visible.

The **close** method posts an NSWindowWillCloseNotification to the default notification center.

The **close** method differs in two important ways from the **performClose:** method:

- It does not attempt to send a **windowShouldClose:** message to the receiver or its delegate.
- It does not simulate the user clicking the close button by momentarily highlighting the button.

Use **performClose:** if you need these features.

See also: – **orderOut:**

constrainFrameRect:toScreen:

– (NSRect)**constrainFrameRect:**(NSRect)*frameRect* **toScreen:**(NSScreen *)*aScreen*

Modifies and returns *frameRect* so that its top edge lies on *aScreen*. If the receiver is resizable, the rectangle's height is adjusted to bring the bottom edge onto the screen as well. The rectangle's width and horizontal location are unaffected. You shouldn't need to invoke this method yourself; it's invoked automatically (and the modified frame is used to locate and set the size of the receiver) whenever a titled NSWindow is placed on screen and whenever its size is changed.

Subclasses can override this method to prevent their instances from being constrained or to constrain them differently.

contentView

– (id)**contentView**

Returns the receivers's content view, the highest accessible NSView object in the receiver's view hierarchy.

See also: – **setContentView:**

convertBaseToScreen:

– (NSPoint)**convertBaseToScreen:**(NSPoint)*aPoint*

Converts *aPoint* from the receiver's base coordinate system to the screen coordinate system. Returns the converted point.

See also: – **convertScreenToBase:**, – **convertPoint:toView:** (NSView)

convertScreenToBase:

– (NSPoint)**convertScreenToBase:**(NSPoint)*aPoint*

Converts *aPoint* from the screen coordinate system to the receiver's base coordinate system. Returns the converted point.

See also: – **convertBaseToScreen:**, – **convertPoint:fromView:** (NSView)

currentEvent

– (NSEvent *)**currentEvent**

Returns the event currently being processed by the application, by invoking NSApplication's **currentEvent** method.

dataWithEPSInsideRect:

– (NSData *)**dataWithEPSInsideRect:**(NSRect)*aRect*

Returns EPS data that draws the region of the receiver within *aRect* (which is expressed in the receiver's base coordinate system). This data can be placed on a pasteboard, written to a file, or used to create an NSImage object.

See also: – **dataWithEPSInsideRect:** (NSView), – **writeEPSInsideRect:toPasteboard:** (NSView)

deepestScreen

– (NSScreen *)**deepestScreen**

Returns the deepest screen that the receiver is on (it may be split over several screens), or **nil** if the receiver is off screen.

See also: – **screen**

defaultButtonCell

– (NSButtonCell *)**defaultButtonCell**

Returns the button cell that performs as if clicked when the NSWindow receives a Return (or Enter) key event. This cell draws itself as the focal element for keyboard interface control, unless another button cell is focused on, in which case the default button cell temporarily draws itself as normal and disables its key equivalent.

The window receives a Return key event if no responder in its responder chain claims it, or if the user presses the Control key along with the Return key.

See also: – **setDefaultButtonCell:**, – **disableKeyEquivalentForDefaultButtonCell**, – **enableKeyEquivalentForDefaultButtonCell**

delegate

– (id)**delegate**

Returns the receiver's delegate, or returns **nil** if it doesn't have one.

See also: – **setDelegate:**

deminiaturize:

– (void)**deminiaturize:**(id)*sender*

Deminiaturizes the receiver. You rarely need to invoke this method; it's invoked automatically when an NSWindow is deminiaturized by the user.

See also: – **miniaturize:**, – **styleMask**

depthLimit

– (NSWindowDepth)**depthLimit**

Returns the depth limit of the receiver. The value returned can be examined with the Application Kit functions **NSPlanarFromDepth**, **NSColorSpaceFromDepth**, **NSBitsPerSampleFromDepth**, **NSBitsPerPixelFromDepth**.

See also: + **defaultDepthLimit**, – **setDepthLimit:**, – **setDynamicDepthLimit:**

deviceDescription

– (NSDictionary *)**deviceDescription**

Returns a dictionary containing information about the receiver's resolution, color depth, and so on. This information is useful for tuning images and colors to the window's display capabilities. The contents of the dictionary are:

Dictionary Key	Value
NSDeviceResolution	An NSValue containing a value of type NSSize that describe the receiver's raster resolution in dots per inch (dpi).
NSDeviceColorSpaceName	An NSString giving the name of the receiver's color space. See the Application Kit Types and Constants for a list of possible values.
NSDeviceBitsPerSample	An NSNumber containing an integer that gives the bit depth of the receiver's raster image (2-bit, 8-bit, and so forth).
NSDeviceIsScreen	YES, indicating that the receiver displays on the screen.
NSDeviceSize	An NSValue containing a value of type NSSize that gives the size of the receiver's frame rectangle.

See also: – **deviceDescription** (NSScreen), – **bestRepresentationForDevice:** (NSImage), – **colorUsingColorSpaceName:** (NSColor)

disableCursorRects

– (void)**disableCursorRects**

Disables all cursor rectangle management within the receiver. Use this method when you need to do some special cursor manipulation and you don't want the Application Kit interfering.

See also: – **enableCursorRects**

disableFlushWindow

– (void)**disableFlushWindow**

Disables the **flushWindow** method for the receiver. If the receiver is buffered, disabling **flushWindow** prevents drawing from being automatically flushed by NSView's **display...** methods from the receiver's backing store to the screen. This permits several NSViews to be drawn before the results are shown to the user.

Flushing should be disabled only temporarily, while the NSWindow's display is being updated. Each **disableFlushWindow** message must be paired with a subsequent **enableFlushWindow** message. Invocations of these methods can be nested; flushing isn't reenabled until the last (unnested) **enableFlushWindow** message is sent.

disableKeyEquivalentForDefaultButtonCell

– (void)**disableKeyEquivalentForDefaultButtonCell**

Disables the default button cell's key equivalent, so that it doesn't perform a click when the user presses Return (or Enter). See the method description for **defaultButtonCell** for more information.

See also: – **enableKeyEquivalentForDefaultButtonCell**

discardCachedImage

– (void)**discardCachedImage**

Discards all of the receiver's cached image rectangles. An NSWindow automatically discards its cached image rectangles when it displays.

See also: – **cacheImageInRect:**, – **restoreCachedImage**, – **display**

discardCursorRects

– (void)**discardCursorRects**

Invalidates all cursor rectangles in the receiver. This method is invoked by **resetCursorRects** to clear out existing cursor rectangles before resetting them. You shouldn't invoke it in the code you write, but might want to override it to change its behavior.

discardEventsMatchingMask:beforeEvent:

– (void)**discardEventsMatchingMask:**(unsigned int)*mask* **beforeEvent:**(NSEvent *)*lastEvent*

Forwards the message to the NSApplication object, which handles it as described in the NSApplication class specification.

display

– (void)**display**

Passes a **display** message down the receiver's view hierarchy, thus redrawing all NSViews within the receiver, including the frame view which draws the border, title bar, and other peripheral elements.

You rarely need to invoke this method. NSWindows normally record which of their NSViews need display and display them automatically on each pass through the event loop.

See also: – **display** (NSView), – **displayIfNeeded**, – **isAutodisplay**

displayIfNeeded

– (void)**displayIfNeeded**

Passes a **displayIfNeeded** message down the receiver's view hierarchy, thus redrawing all NSViews that need to be displayed, including the frame view which draws the border, title bar, and other peripheral elements. This method is useful when you want to modify some number of NSViews and then display only the ones that were modified.

You rarely need to invoke this method. NSWindows normally record which of their NSViews need display and display them automatically on each pass through the event loop.

See also: – **display**, – **displayIfNeeded** (NSView), – **setNeedsDisplay:** (NSView), – **isAutodisplay**

dragImage:at:offset:event:pasteboard:source:slideBack:

– (void)**dragImage:**(NSImage *)*anImage*
at:(NSPoint)*aPoint*
offset:(NSSize)*initialOffset*
event:(NSEvent *)*theEvent*
pasteboard:(NSPasteboard *)*pboard*
source:(id)*sourceObject*
slideBack:(BOOL)*flag*

Begins a dragging session. This method should be invoked only from within a view's implementation of the **mouseDown:** method (which overrides the version defined in NSResponder). Essentially the same as NSView's method of the same name, except that *aPoint* is given in the NSWindow's base coordinate system. See the description of this method in the NSView class specification for more information.

enableCursorRects

– (void)**enableCursorRects**

Reenables cursor rectangle management within the receiver after a **disableCursorRects** message.

enableFlushWindow

– (void)**enableFlushWindow**

Reenables the **flushWindow** method for the receiver after it was disabled through a previous **disableFlushWindow** message.

enableKeyEquivalentForDefaultButtonCell

– (void)**enableKeyEquivalentForDefaultButtonCell**

Reenables the default button cell's key equivalent, so that it performs a click when the user presses Return (or Enter). See the method description for **defaultButtonCell** for more information.

See also: – **disableKeyEquivalentForDefaultButtonCell**

endEditingFor:

– (void)**endEditingFor:**(id)*anObject*

Forces the field editor, which *anObject* is assumed to be using, to give up its first responder status, and prepares it for its next assignment. If the field editor is the first responder, it's made to resign that status even if its **resignFirstResponder** method returns NO. This forces the field editor to send a **textDidEndEditing:** message to its delegate. The field editor is then removed from the view hierarchy, its delegate is set to **nil**, and it's emptied of any text it may contain.

This method is typically invoked by the object using the field editor when it's finished. Other objects normally change the first responder by simply using **makeFirstResponder:**, which allows a field editor or other object to retain its first responder status if, for example, the user has entered an invalid value. The **endEditingFor:** method should be used only as a last resort if the field editor refuses to resign first responder status. Even in this case, you should always allow the field editor a chance to validate its text and take whatever other action it needs first. You can do this by first trying to make the NSWindow the first responder:

```
if ([myWindow makeFirstResponder:myWindow]) {
    /* All fields are now valid; it's safe to use fieldEditor:forObject:
       to claim the field editor. */
}
else {
    /* Force first responder to resign. */
    [myWindow endEditingFor:nil];
}
```

See also: – **fieldEditor:forObject:**, – **windowWillReturnFieldEditor:toObject:**

fax:

– (void)**fax:**(id)*sender*

Runs the Fax panel, and if the user chooses an option other than canceling, prints the receiver (its frame view and all subviews) to a fax modem.

See also: – **print:**

fieldEditor:forObject:

– (NSText *)**fieldEditor:**(BOOL)*createFlag* **forObject:**(id)*anObject*

Returns the receiver's field editor, creating it if needed and if *createFlag* is YES. Returns **nil** if *createFlag* is NO and the field editor doesn't exist. *anObject* is used to allow the receiver's delegate to substitute another object in place of the field editor, as described below. The field editor may be in use by some view object, so be sure to properly dissociate it from that object before actually using it yourself (the appropriate way to do this is illustrated in the description of **endEditingFor:**). Once you retrieve the field editor, you can insert it in the view hierarchy, set a delegate to interpret text events, and have it perform whatever editing is needed. Then, when it sends a **textDidEndEditing:** message to the delegate, you can get its text to display or store, and remove the field editor using **endEditingFor:**.

The field editor is provided as a convenience and can be used however your application sees fit. Typically, the field editor is used by simple text-bearing objects—for example, an NSTextField object uses its window's field editor to display and manipulate text. The field editor can be shared by any number of objects and so its state may be constantly changing. Therefore, it shouldn't be used to display text that demands sophisticated layout (for this you should create a dedicated NSText object).

A freshly created NSWindow doesn't have a field editor. After a field editor has been created for an NSWindow, the *createFlag* argument is ignored. By passing NO for *createFlag* and testing the return value, however, you can predicate an action on the existence of the field editor.

The receiver's delegate can substitute a custom editor in place of the NSWindow's field editor by implementing **windowWillReturnFieldEditor:toObject:**. The receiver sends this message to its delegate with itself and *anObject* as the arguments, and if the return value is not **nil** the NSWindow returns that object instead of its field editor. However, note the following:

- If the NSWindow's delegate is identical to *anObject*, **windowWillReturnFieldEditor:toObject:** isn't sent.
- The object returned by the delegate method, though it may become first responder, does *not* become the NSWindow's field editor. Other objects continue to use the NSWindow's established field editor.

firstResponder

– (NSResponder *)**firstResponder**

Returns the receiver's first responder.

See also: – **makeFirstResponder:**, – **acceptsFirstResponder** (NSResponder)

flushWindow

– (void)**flushWindow**

Flushes the receiver's off-screen buffer to the screen if the receiver is buffered and flushing is enabled. Does nothing for other display devices, such as a printer. This method is automatically invoked by NSWindow's and NSView's **display** and **displayIfNeeded** methods.

See also: – **flushWindowIfNeeded**, – **disableFlushWindow**, – **enableFlushWindow**

flushWindowIfNeeded

– (void)**flushWindowIfNeeded**

Flushes the receiver's off-screen buffer to the screen if flushing is enabled and if the last **flushWindow** message had no effect because flushing was disabled. To avoid unnecessary flushing, use this method rather than **flushWindow** to flush an NSWindow after flushing has been reenabled.

See also: – **flushWindow**, – **disableFlushWindow**, – **enableFlushWindow**

frame

– (NSRect)**frame**

Returns the receiver's frame rectangle. The frame rectangle is always reckoned in the screen coordinate system.

See also: – **screen**, – **deepestScreen**

frameAutosaveName

– (NSString *)**frameAutosaveName**

Returns the name used to automatically save the receiver's frame rectangle data in the defaults system, as set through **setFrameAutosaveName:**. If the receiver has an autosave name, its frame data is written whenever the frame rectangle changes.

See also: – **setFrameUsingName:**

gState

– (int)**gState**

Returns the PostScript graphics state object associated with the receiver. This graphics state is used by default for all NSViews in the receiver's view hierarchy, but individual NSViews can be made to use their own with the NSView method **allocateGState**.

hasDynamicDepthLimit

– (BOOL)**hasDynamicDepthLimit**

Returns YES if the receiver's depth limit can change to match the depth of the screen it's on, NO if it can't.

See also: – **setDynamicDepthLimit:**

hidesOnDeactivate

– (BOOL)**hidesOnDeactivate**

Returns YES if the receiver is removed from the screen when its application is deactivated, NO if it remains on screen.

See also: – **setHidesOnDeactivate:**

initialFirstResponder

– (NSView *)**initialFirstResponder**

Returns the NSView that's made first responder the first time the receiver is placed on screen.

See also: – **setInitialFirstResponder:**, – **setNextKeyView:** (NSView)

initWithContentRect:styleMask:backing:defer:

– (id)**initWithContentRect:**(NSRect)*contentRect*
styleMask:(unsigned int)*styleMask*
backing:(NSBackingStoreType)*backingType*
defer:(BOOL)*flag*

Initializes the receiver, a newly allocated NSWindow object, and returns **self**. This method is the designated initializer for the NSWindow class.

contentRect specifies the location and size of the NSWindow's content area in screen coordinates. Note that the Window Server limits window position coordinates to ±16,000 and sizes to 10,000.

styleMask specifies the receiver's style. It can either be NSBorderlessWindowMask, or it can contain any of the following options, combined using the C bitwise OR operator:

Option	Meaning
NSTitledWindowMask	The NSWindow displays a title bar.
NSClosableWindowMask	The NSWindow displays a close button.
NSMiniaturizableWindowMask	The NSWindow displays a miniaturize button.
NSResizableWindowMask	The NSWindow displays a resize bar or border.

Borderless windows display none of the usual peripheral elements and are generally useful only for display or caching purposes; you should normally not need to create them. Also, note that an NSWindow's style mask should include NSTitledWindowMask if it includes any of the others.

backingType specifies how the drawing done in the receiver is buffered by the object's window device:

NSBackingStoreBuffered
NSBackingStoreRetained
NSBackingStoreNonretained

flag determines whether the Window Server creates a window device for the new object immediately. If *flag* is YES, it defers creating the window until the receiver is moved on screen. All display messages sent to the NSWindow or its NSViews are postponed until the window is created, just before it's moved on screen. Deferring the creation of the window improves launch time and minimizes the virtual memory load on the Window Server.

The new NSWindow creates an instance of NSView to be its default content view. You can replace it with your own object by using the **setContentView:** method.

See also: – **orderFront:**, – **setTitle:**, – **setOneShot:**, – **initWithContentRect:styleMask:backing:defer:screen:**

initWithContentRect:styleMask:backing:defer:screen:

– (id)**initWithContentRect:**(NSRect)*contentRect*
styleMask:(unsigned int)*styleMask*
backing:(NSBackingStoreType)*bufferingType*
defer:(BOOL)*flag*
screen:(NSScreen *)*aScreen*

Initializes a newly allocated NSWindow object and returns **self**. This method is equivalent to **initWithContentRect:styleMask:backing:defer:**, except that the content rectangle is specified relative to the lower-left corner of *aScreen*.

If *aScreen* is **nil**, the content rectangle is interpreted relative to the lower-left corner of the main screen. The main screen is the one that contains the current key window, or, if there is no key window, the one that contains the main menu. If there's neither a key window nor a main menu (if there's no active application), the main screen is the one where the origin of the screen coordinate system is located.

See also: – **orderFront:**, – **setTitle:**, – **setOneShot:**

interfaceStyle

– (NSInterfaceStyle)**interfaceStyle**

Returns the receiver's interface style, such as NSMacintoshInterfaceStyle or NSWindows95InterfaceStyle. A responder's style (if other than NSNoInterfaceStyle) overrides all other settings, such as those established by the defaults system.

See also: – **setInterfaceStyle:**

invalidateCursorRectsForView:

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

Marks as invalid the cursor rectangles of *aView*, an NSView in the receiver's view hierarchy, so that they'll be set up again when the receiver becomes key (or immediately if the receiver is key).

See also: – **resetCursorRects**, – **resetCursorRects** (NSView)

isAutodisplay

– (BOOL)**isAutodisplay**

Returns YES if the receiver automatically displays its views that are marked as needing it, NO if it doesn't. Automatic display typically occurs on each pass through the event loop.

See also: – **setAutodisplay:**, – **displayIfNeeded**, – **setNeedsDisplay:** (NSView)

isDocumentEdited

– (BOOL)**isDocumentEdited**

Returns YES or NO according to the argument supplied with the last **setDocumentEdited:** message.

isExcludedFromWindowsMenu

– (BOOL)**isExcludedFromWindowsMenu**

Returns YES if the receiver's title is omitted from the application's Windows menu, NO if it is listed.

See also: – **setExcludedFromWindowsMenu:**

isFlushWindowDisabled

– (BOOL)**isFlushWindowDisabled**

Returns YES if the receiver's flushing ability has been disabled; otherwise returns NO.

See also: – **disableFlushWindow**, – **enableFlushWindow**

isKeyWindow

– (BOOL)**isKeyWindow**

Returns YES if the receiver is the key window for the application, NO if it isn't.

See also: – **isMainWindow**, – **makeKeyWindow**

isMainWindow

– (BOOL)**isMainWindow**

Returns YES if the receiver is the main window for the application, NO if it isn't.

See also: – **isKeyWindow**, – **makeMainWindow**

isMiniaturized

– (BOOL)**isMiniaturized**

Returns YES if the receiver has been miniaturized, NO if it hasn't. A miniaturized window is removed from the screen and replaced by a miniwindow, icon, or button that represents it, called the *counterpart* (the particular form depends on the platform).

See also: – **miniaturize:**

isOneShot

– (BOOL)**isOneShot**

Returns YES if the PostScript window device that the receiver manages is freed when it's removed from the screen list, NO if not. The default is NO.

See also: – **setOneShot:**

isReleasedWhenClosed

– (BOOL)**isReleasedWhenClosed**

Returns YES if the receiver is automatically released after being closed, NO if it's simply removed from the screen. The default for NSWindow is YES; the default for NSPanel is NO.

See also: – **setReleasedWhenClosed:**

isVisible

– (BOOL)**isVisible**

Returns YES if the receiver is on screen (even if it's obscured by other windows).

See also: – **visibleRect** (NSView)

keyDown:

– (void)**keyDown:**(NSEvent *)*theEvent*

Handles a keyboard event that may need to be interpreted as changing the key view or triggering a mnemonic.

See also: – **selectNextKeyView:**, – **nextKeyView** (NSView), – **performMnemonic:** (NSView)

keyViewSelectionDirection

– (NSSelectionDirection)**keyViewSelectionDirection**

Returns the direction that the receiver is currently using to change the key view, one of:

Value	Meaning
NSDirectSelection	The receiver isn't traversing the key view loop.
NSSelectingNext	The receiver is proceeding to the next valid key view.
NSSelectingPrevious	The receiver is proceeding to the previous valid key view.

See also: – **selectNextKeyView:**, – **selectPreviousKeyView:**

level

– (int)**level**

Returns the level of the receiver as set using **setLevel:**. See that method description for a list of possible values.

makeFirstResponder:

– (BOOL)**makeFirstResponder:**(NSResponder *)*aResponder*

Attempts to make *aResponder* the first responder for the receiver. If *aResponder* isn't already the first responder, this method first sends a **resignFirstResponder** message to the object that is. If that object refuses to resign, it remains the first responder and this method immediately returns NO. If it returns YES, this methods sends a **becomeFirstResponder** message to *aResponder*. If *aResponder* accepts first responder status, this method returns YES. If it refuses, this method returns NO, and the NSWindow becomes first responder.

The Application Kit uses this method to alter the first responder in response to mouse-down events; you can also use it to explicitly set the first responder from within your program. *aResponder* is typically an NSView in the receiver's view hierarchy.

See also: – **becomeFirstResponder** (NSResponder), – **resignFirstResponder** (NSResponder)

makeKeyAndOrderFront:

– (void)**makeKeyAndOrderFront:**(id)*sender*

Moves the receiver to the front of the screen list, within its level, and makes it the key window.

See also: – **orderFront:**, – **orderBack:**, – **orderOut:**, – **orderWindow:relativeTo:**, – **setLevel:**

makeKeyWindow

– (void)**makeKeyWindow**

Makes the receiver the key window.

See also: – **makeMainWindow**, – **becomeKeyWindow**, – **isKeyWindow**

makeMainWindow

– (void)**makeMainWindow**

Makes the receiver the main window.

See also: – **makeKeyWindow**, – **becomeMainWindow**, – **isMainWindow**

maxSize

– (NSSize)**maxSize**

Returns the maximum size to which the receiver's frame can be sized either by the user or by the **setFrame...** methods other than **setFrame:display:**.

See also: – **setMaxSize:**, – **minSize**, – **aspectRatio**, – **resizeIncrements**

miniaturize:

– (void)**miniaturize:**(id)*sender*

Removes the receiver from the screen list and displays its counterpart in the appropriate location.

See also: – **deminiaturize:**

miniwindowImage

– (NSImage *)**miniwindowImage**

Returns the image that's displayed in the receiver's miniwindow.

See also: – **setMiniwindowImage:**, – **miniwindowTitle**

miniwindowTitle

– (NSString *)**miniwindowTitle**

Returns the title that's displayed in the receiver's miniwindow.

See also: – **setMiniwindowTitle:**, – **miniwindowImage**

minSize

– (NSSize)**minSize**

Returns the minimum size to which the receiver's frame can be sized either by the user or by the **setFrame...** methods other than **setFrame:display:**.

See also: – **setMinSize:**, – **maxSize**, – **aspectRatio**, – **resizeIncrements**

mouseLocationOutsideOfEventStream

– (NSPoint)**mouseLocationOutsideOfEventStream**

Returns the current location of the mouse reckoned in the receiver's base coordinate system, regardless of the current event being handled or of any events pending.

See also: – **currentEvent** (NSApplication)

nextEventMatchingMask:

– (NSEvent *)**nextEventMatchingMask:**(unsigned int)*mask*

Invokes NSApplication's **nextEventMatchingMask:untilDate:inMode:dequeue:** method, using *mask* as the first argument, with an unlimited expiration, a mode of NSEventTrackingRunLoopMode, and a dequeue flag of YES. See the method description in the NSApplication class specification for more information.

nextEventMatchingMask:untilDate:inMode:dequeue:

– (NSEvent *)**nextEventMatchingMask:**(unsigned int)*mask*
untilDate:(NSDate *)*expirationDate*
inMode:(NSString *)*mode*
dequeue:(BOOL)*flag*

Forwards the message to the global NSApplication object, NSApp. See the method description in the NSApplication class specification for more information.

orderBack:

– (void)**orderBack:**(id)*sender*

Moves the receiver to the back of its level in the screen list, without changing either the key window or the main window.

See also: – **orderFront:**, – **orderOut:**, – **orderWindow:relativeTo:**, – **makeKeyAndOrderFront:**, – **level**

orderFront:

– (void)**orderFront:**(id)*sender*

Moves the receiver to the front of its level in the screen list, without changing either the key window or the main window.

See also: – **orderBack:**, – **orderOut:**, – **orderWindow:relativeTo:**, – **makeKeyAndOrderFront:**, – **level**

orderFrontRegardless

– (void)**orderFrontRegardless**

Moves the receiver to the front of its level, even if its application isn't active, but without changing either the key window or the main window. Normally an NSWindow can't be moved in front of the key window unless the NSWindow and the key window are in the same application. You should rarely need to invoke this method; it's designed to be used when applications are cooperating in such a way that an active application (with the key window) is using another application to display data.

See also: – **orderFront:**, – **level**

orderOut:

– (void)**orderOut:**(id)*sender*

Takes the receiver out of the screen list. If the receiver is the key or main window, the NSWindow immediately behind it is made key or main in its place. Calling the **orderOut:** method causes the receiver to be removed from the screen, but does not cause it to be released. See the **close** method for information on when a window is released.

See also: – **orderFront:**, – **orderBack:**, – **orderWindow:relativeTo:**, – **setReleasedWhenClosed:**

orderWindow:relativeTo:

– (void)**orderWindow:**(NSWindowOrderingMode)*place* **relativeTo:**(int)*otherWindowNumber*

Repositions the receiver's window device in the Window Server's screen list. If *place* is NSWindowOut, the receiver is removed from the screen list and *otherWindowNumber* is ignored. If it's NSWindowAbove the receiver is ordered immediately in front of the window whose window number is *otherWindowNumber*. Similarly, if *place* is NSWindowBelow, the receiver is placed immediately behind the window represented by *otherWindowNumber*. If *otherWindowNumber* is 0, the receiver is placed in front of or behind all other windows in its level.

See also: – **orderFront:**, – **orderBack:**, – **orderOut:**, – **makeKeyAndOrderFront:**, – **level**, – **windowNumber**

performClose:

– (void)**performClose:**(id)*sender*

Simulates the user clicking the close button by momentarily highlighting the button and then closing the window. If the receiver's delegate or the receiver itself implements **windowShouldClose:**, then that message is sent with the receiver as the argument. (Only one such message is sent; if both the delegate and the NSWindow implement the method, only the delegate receives the message.) If the **windowShouldClose:** method returns NO, the window isn't closed. If it returns YES, or if it isn't implemented, **performClose:** invokes the **close** method to close the window.

If the receiver doesn't have a close button or can't be closed (for example, if the delegate replies NO to a **windowShouldClose:** message), then this method calls the **NSBeep** function.

See also: – **styleMask**, – **performClick:** (NSButton), – **performMiniaturize:**

performMiniaturize:

– (void)**performMiniaturize:**(id)*sender*

Simulates the user clicking the miniaturize button by momentarily highlighting the button then miniaturizing the window. If the receiver doesn't have a miniaturize button or can't be miniaturized for some reason, this method calls the **NSBeep** function.

See also: – **close**, – **styleMask**, – **performClick:** (NSButton), – **performClose:**

performZoom:

– (void)**performZoom:**(id)*sender*

Simulates the user clicking the zoom box by momentarily highlighting the button and then zooming the window. If the receiver doesn't have a zoom box or can't be zoomed for some reason, this method calls the **NSBeep** function.

See also: – **styleMask**, – **performClick:** (NSButton), – **zoom:**

postEvent:atStart:

– (void)**postEvent:**(NSEvent *)*anEvent* **atStart:**(BOOL)*flag*

Forwards the message to the global NSApplication object, NSApp.

print:

– (void)**print:**(id)*sender*

Runs the Print panel, and if the user chooses an option other than canceling, prints the receiver (its frame view and all subviews).

See also: – **fax:**

registerForDraggedTypes:

– (void)**registerForDraggedTypes:**(NSArray *)*pboardTypes*

Registers *pboardTypes* as the pasteboard types that the receiver will accept as the destination of an image-dragging session.

Note: Registering an NSWindow for dragged types automatically makes it a candidate destination object for a dragging session. As such, it must properly implement some or all of the NSDraggingDestination protocol methods. As a convenience, NSWindow provides default

implementations of these methods. See the NSDraggingDestination protocol specification for details.

See also: – **unregisterDraggedTypes**

representedFilename

– (NSString *)**representedFilename**

Returns the name of the file that the receiver represents.

See also: – **setRepresentedFilename:**

resetCursorRects

– (void)**resetCursorRects**

Invokes **discardCursorRects** to clear the receiver's cursor rectangles, then sends **resetCursorRects** to every NSView in the receiver's view hierarchy.

This method is typically invoked by the NSApplication object when it detects that the key window's cursor rectangles are invalid. In program code, it's more efficient to invoke **invalidateCursorRectsForView:**.

resignKeyWindow

– (void)**resignKeyWindow**

Never invoke this method; it's invoked automatically when the NSWindow resigns key window status. This method sends **resignKeyWindow** to the receiver's first responder, sends **windowDidResignKey:** to the receiver's delegate, and posts an NSWindowDidResignKeyNotification to the default notification center.

See also: – **becomeKeyWindow**, – **resignMainWindow**

resignMainWindow

– (void)**resignMainWindow**

Never invoke this method; it's invoked automatically when the NSWindow resigns main window status. This method sends **windowDidResignMain:** to the receiver's delegate and posts an NSWindowDidResignMainNotification to the default notification center.

See also: – **becomeMainWindow**, – **resignKeyWindow**

resizeFlags

– (int)**resizeFlags**

Valid only while the receiver is being resized, this method returns the flags field of the event record for the mouse-down event that initiated the resizing session. The integer encodes, as a mask, which of the modifier keys was held down when the event occurred. The flags are listed in the NSEvent class's **modifierFlags** method description. You can use this method to constrain the direction or amount of resizing. Because of its limited validity, this method should only be invoked from within an implementation of the delegate method **windowWillResize:toSize:**.

resizeIncrements

– (NSSize)**resizeIncrements**

Returns the receiver's resizing increments, which restrict the user's ability to resize it so that its width and height alter by integral multiples of *increments.width* and *increments.height* when the user resizes it. These amounts are whole number values, 1.0 or greater. You can set an NSWindow's size to any value programmatically.

See also: – **setResizeIncrements:**, – **setAspectRatio:**, – **setFrame:display:**

restoreCachedImage

– (void)**restoreCachedImage**

Splices the receiver's cached image rectangles, if any, back into its raster image (and buffer if it has one), undoing the effect of any drawing performed within those areas since they were established using **cacheImageInRect:**. You must invoke **flushWindowIfNeeded** after this method to guarantee proper redisplay. An NSWindow automatically discards its cached image rectangles when it displays.

See also: – **discardCachedImage**, – **display**

saveFrameUsingName:

– (void)**saveFrameUsingName:**(NSString *)*name*

Saves the receiver's frame rectangle in the user defaults system. With the companion method **setFrameUsingName:**, you can save and reset an NSWindow's frame over various launchings of an application. The default is owned by the application and stored under the name "NSWindow Frame *name*". See the NSUserDefaults class specification for more information.

See also: – **stringWithSavedFrame**

screen

– (NSScreen *)**screen**

Returns the screen that the receiver is on. If the receiver is partly on one screen and partly on another, the screen where most of it lies is the one returned.

See also: – **deepestScreen**

selectKeyViewFollowingView:

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

Sends the NSView message **nextValidKeyView:** to *aView*, and if that message returns an NSView, invokes **makeFirstResponder:** with the returned NSView.

See also: – **selectKeyViewPrecedingView:**

selectKeyViewPrecedingView:

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

Sends the NSView message **previousValidKeyView:** to *aView*, and if that message returns an NSView, invokes **makeFirstResponder:** with the returned NSView.

See also: – **selectKeyViewFollowingView:**

selectNextKeyView:

– (void)**selectNextKeyView:**(id)*sender*

Searches for a candidate key view and, if it finds one, invokes **makeFirstResponder:** to establish it as the first responder. The candidate is one of the following (searched for in this order):

- The current first responder's next valid key view, as returned by NSView's **nextValidKeyView:** method.
- The object designated as the receiver's initial first responder (using **setInitialFirstResponder:**) if it returns YES to an **acceptsFirstResponder** message.
- Otherwise, the initial first responder's next valid key view, which may end up being **nil**.

See also: – **selectPreviousKeyView:**, – **selectKeyViewFollowingView:**

selectPreviousKeyView:

– (void)**selectPreviousKeyView:**(id)*sender*

Searches for a candidate key view and, if it finds one, invokes **makeFirstResponder:** to establish it as the first responder. The candidate is one of the following (searched for in this order):

- The current first responder's previous valid key view, as returned by NSView's **previousValidKeyView:** method.
- The object designated as the receiver's initial first responder (using **setInitialFirstResponder:**) if it returns YES to an **acceptsFirstResponder** message.
- Otherwise, the initial first responder's previous valid key view, which may end up being **nil**.

See also: – **selectNextKeyView:**, – **selectKeyViewPrecedingView:**

sendEvent:

– (void)**sendEvent:**(NSEvent *)*theEvent*

Dispatches mouse and keyboard events sent to the receiver by the NSApplication object. Never invoke this method directly.

setAcceptsMouseMovedEvents:

– (void)**setAcceptsMouseMovedEvents:**(BOOL)*flag*

Sets whether the receiver accepts mouse-moved events and distributes them to its responders. If *flag* is YES it does accept them; if *flag* is NO it doesn't. NSWindows by default don't accept mouse-moved events.

See also: – **acceptsMouseMovedEvents**

setAspectRatio:

– (void)**setAspectRatio:**(NSSize)*ratio*

Sets the receiver's size aspect ratio to *ratio*, constraining the size of its frame rectangle to integral multiples of this size when the user resizes it.

An NSWindow's aspect ratio and its resize increments are mutually exclusive attributes. In fact, setting one attribute cancels the setting of the other. For example, to cancel an established aspect ratio setting for an NSWindow, you send the NSWindow object a **setResizeIncrements:** message with the width and height set to 1.0:

```
[myWindow setResizeIncrements:NSMakeSize(1.0,1.0)];
```

See also: – **aspectRatio**, – **setFrame:display:**

setAutodisplay:

– (void)**setAutodisplay:**(BOOL)*flag*

Sets whether the receiver automatically displays its views that are marked as needing it. If *flag* is YES, views are automatically displayed as needed, typically on each pass through the event loop. If *flag* is NO, the receiver or its views must be explicitly displayed.

See also: – **isAutodisplay**, – **displayIfNeeded**, – **displayIfNeeded** (NSView)

setBackgroundColor:

– (void)**setBackgroundColor:**(NSColor *)*aColor*

Sets the color of the receiver's background to *aColor*.

See also: – **backgroundColor**

setBackingType:

– (void)**setBackingType:**(NSBackingStoreType)*backingType*

Sets the receiver's backing store type to *backingType*, which may be one of the following constants:

NSBackingStoreBuffered
NSBackingStoreRetained

This method can only be used to switch a buffered NSWindow to retained or vice versa; you can't change the backing type to or from nonretained after initializing an NSWindow (a PostScript error is generated if you attempt to do so).

See also: – **backingType**, **– initWithContentRect:styleMask:backing:defer:**, **– initWithContentRect:styleMask:backing:defer:screen:**

setContentSize:

– (void)**setContentSize:**(NSSize)*aSize*

Sets the size of the receiver's content view to *aSize*, which is expressed in the receiver's base coordinate system. This in turn alters the size of the NSWindow itself. Note that the Window Server limits window sizes to 10,000; if necessary, be sure to limit *aSize* relative to the frame rectangle.

See also: – **setFrame:display:**, + **contentRectForFrameRect:styleMask:**, + **frameRectForContentRect:styleMask:**

setContentView:

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

Makes *aView* the receiver's content view; the previous content view is removed from the receiver's view hierarchy and released. *aView* is resized to fit precisely within the content area of the NSWindow. You can modify the content view's coordinate system through its bounds rectangle, but can't alter its frame rectangle (that is, its size or location) directly.

This method causes the old content view to be released; if you plan to reuse it, be sure to retain it before sending this message and to release it as appropriate when adding it to another NSWindow or NSView.

See also: – **contentView**, – **setContentSize:**

setDefaultButtonCell:

– (void)**setDefaultButtonCell:**(NSButtonCell *)*aButtonCell*

Makes the key equivalent of *aButtonCell*'s the Return (or Enter) key, so that when the user presses Return that button performs as if clicked. See the method description for **defaultButtonCell** for more information.

See also: – **disableKeyEquivalentForDefaultButtonCell**,
– **enableKeyEquivalentForDefaultButtonCell**

setDelegate:

– (void)**setDelegate:**(id)*anObject*

Makes *anObject* the receiver's delegate, without retaining it. An NSWindow's delegate is inserted in the responder chain after the NSWindow itself and is informed of various actions by the NSWindow through delegation messages.

See also: – **delegate**, – **tryToPerform:with:**, – **sendAction:to:from:** (NSApplication)

setDepthLimit:

– (void)**setDepthLimit:**(NSWindowDepth)*limit*

Sets the depth limit of the receiver to *limit*, which can be determined using the **NSBestDepth** function. Passing a value of 0 for *limit* sets the depth limit to the receiver's default depth limit; using a value of 0 can be useful for reverting an NSWindow to its initial depth.

See also: – **depthLimit**, + **defaultDepthLimit**, – **setDynamicDepthLimit:**

setDocumentEdited:

– (void)**setDocumentEdited:**(BOOL)*flag*

Sets whether the receiver's document has been edited and not saved. NSWindows are by default in "not edited" status.

You should invoke this method with an argument of YES every time the NSWindow's document changes in such a way that it needs to be saved and with an argument of NO every time it gets saved. Then, before closing the NSWindow you can use **isDocumentEdited** to determine whether to allow the user a chance to save the document.

setDynamicDepthLimit:

– (void)**setDynamicDepthLimit:**(BOOL)*flag*

Sets whether the receiver changes its depth to match the depth of the screen that it's on, or the depth of the deepest screen when it spans multiple screens. If *flag* is YES, the depth limit depends on which screen the receiver is on. If *flag* is NO, the receiver uses either its preset depth limit or the default depth limit. A different, and nondynamic, depth limit can be set with the **setDepthLimit:** method.

See also: – **hasDynamicDepthLimit**, + **defaultDepthLimit**

setExcludedFromWindowsMenu:

– (void)**setExcludedFromWindowsMenu:**(BOOL)*flag*

Sets whether the receiver's title is omitted from the application's Windows menu. If *flag* is YES it's omitted; if *flag* is NO, it's listed when it or its miniwindow is on screen. The default is NO.

See also: – **isExcludedFromWindowsMenu**

setFrame:display:

– (void)**setFrame:**(NSRect)*frameRect* **display:**(BOOL)*flag*

Sets the origin and size of the receiver's frame rectangle according to *frameRect*, thereby setting its position and size on screen, and invokes **display** if *flag* is YES. Note that the Window Server limits window position coordinates to ±16,000 and sizes to 10,000.

See also: – **frame**, – **setFrameFromString:**, – **setFrameOrigin:**, – **setFrameTopLeftPoint:**, – **setFrameUsingName:**

setFrameAutosaveName:

– (BOOL)**setFrameAutosaveName:**(NSString *)*name*

Sets the name used to automatically save the receiver's frame rectangle in the defaults system to *name*. If *name* isn't the empty string (@""), the receiver's frame is saved as a user default (as described in **saveFrameUsingName:**) each time the frame changes. Returns YES if the name is set successfully, NO if it's being used as an autosave name by another NSWindow in the application (in which case the receiver's old name remains in effect).

See also: + **removeFrameUsingName:**, – **stringWithSavedFrame**, – **setFrameFromString:**

setFrameFromString:

– (void)**setFrameFromString:**(NSString *)*aString*

Sets the receiver's frame rectangle from the string representation *aString*, a representation previously creating using **stringWithSavedFrame**. The frame is constrained according to the receiver's minimum and maximum size settings. This method causes a **windowWillResize:toSize:** message to be sent to the delegate.

setFrameOrigin:

– (void)**setFrameOrigin:**(NSPoint)*aPoint*

Positions the lower-left corner of the receiver's frame rectangle at *aPoint* in screen coordinates. Note that the Window Server limits window position coordinates to ±16,000.

See also: – **setFrame:display:**, – **setFrameTopLeftPoint:**

setFrameTopLeftPoint:

– (void)**setFrameTopLeftPoint:**(NSPoint)*aPoint*

Positions the top-left corner of the receiver's frame rectangle at *aPoint* in screen coordinates. Note that the Window Server limits window position coordinates to ±16,000; if necessary, adjust *aPoint* relative to the window's lower-left corner to account for this.

See also: – **cascadeTopLeftFromPoint:**, – **setFrame:display:**, – **setFrameOrigin:**

setFrameUsingName:

– (BOOL)**setFrameUsingName:**(NSString *)*name*

Sets the receiver's frame rectangle by reading the rectangle data stored in *name* from the defaults system. The frame is constrained according to the receiver's minimum and maximum size settings. This method causes a **windowWillResize:toSize:** message to be sent to the delegate. Returns YES if *name* is read and the frame is set successfully; otherwise returns NO.

See also: – **setFrameAutosaveName:**, + **removeFrameUsingName:**, – **stringWithSavedFrame**, – **setFrameFromString:**

setHidesOnDeactivate:

– (void)**setHidesOnDeactivate:**(BOOL)*flag*

Sets whether the receiver is removed from the screen when the application is inactive. If *flag* is YES, the receiver is hidden (taken out of the screen list) when the application stops being the active application. If *flag* is NO, the receiver stays on screen. The default for NSWindow is NO; the default for NSPanel is YES.

See also: – **hidesOnDeactivate**

setInitialFirstResponder:

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

Sets *aView* as the NSView that's made first responder (also called the key view) the first time the receiver is placed on screen.

See also: – **initialFirstResponder**

setInterfaceStyle:

– (void)**setInterfaceStyle:(**NSInterfaceStyle**)***interfaceStyle*

Sets the receiver's style to the style specified by *interfaceStyle*, such as NSMacintoshInterfaceStyle or NSWindows95InterfaceStyle. You should almost never need to invoke or override this method, but if you do override it, your version should always invoke **super**.

See also: – **interfaceStyle**

setLevel:

– (void)**setLevel:**(int)*newLevel*

Sets the receiver's window level to *newLevel*. Some useful predefined values are:

Level	Comment
NSNormalWindowLevel	The default level for NSWindow objects.
NSFloatingWindowLevel	Useful for floating palettes.
NSDockWindowLevel	Reserved for the application dock (Mach-based systems only).
NSSubmenuWindowLevel	Reserved for submenus (not used on Microsoft Windows). Synonymous with NSTornOffMenuWindowLevel.
NSTornOffMenuWindowLevel	The level for a torn-off menu. Synonymous with NSSubmenuWindowLevel.
NSMainMenuWindowLevel	Reserved for the application's main menu (not used on Microsoft Windows).
NSModalPanelWindowLevel	The level for a modal panel.
NSPopUpMenuWindowLevel	The level for a popup menu.

Each level in the list groups windows within it in front of those in all preceding groups. Floating windows, for example, appear in front of all normal-level windows. When a window enters a new level, it's ordered in front of all of its peers in that level.

The constant NSTornOffMenuWindowLevel is preferable to its synonym, NSSubmenuWindowLevel

See also: – **level**, – **orderWindow:relativeTo:**, – **orderFront:**, – **orderBack:**

setMaxSize:

– (void)**setMaxSize:**(NSSize)*aSize*

Sets the maximum size to which the receiver's frame can be sized to *aSize*. The maximum size constraint is enforced for resizing by the user as well as for the **setFrame...** methods other than **setFrame:display:**. Note that the Window Server limits window sizes to 10,000.

See also: – **maxSize**, – **setMinSize:**, – **setAspectRatio:**, – **setResizeIncrements:**

setMiniwindowImage:

– (void)**setMiniwindowImage:**(NSImage *)*anImage*

Sets the image displayed by the receiver's miniwindow to *anImage*.

See also: – **miniwindowImage**, – **isMiniaturized**

setMiniwindowTitle:

– (void)**setMiniwindowTitle:**(NSString *)*aString*

Sets the title of the receiver's miniaturized counterpart to *aString* and redisplays it. A miniwindow's title normally reflects that of its full-size counterpart, abbreviated to fit if necessary. Although this method allows you to set the miniwindow's title explicitly, changing the full-size NSWindow's title (through **setTitle:** or **setTitleWithRepresentedFilename:**) automatically changes the miniwindow's title as well.

See also: – **miniwindowTitle**

setMinSize:

– (void)**setMinSize:**(NSSize)*aSize*

Returns the minimum size to which the receiver's frame can be sized to *aSize*. The minimum size constraint is enforced for resizing by the user as well as for the **setFrame...** methods other than **setFrame:display:**.

See also: – **minSize**, – **setMaxSize:**, – **setAspectRatio:**, – **setResizeIncrements:**

setOneShot:

– (void)**setOneShot:**(BOOL)*flag*

Sets whether the PostScript window device that the receiver manages should be freed when it's removed from the screen list (and another one created if it's returned to the screen). Freeing the window device when it's removed from the screen list can result in memory savings and performance improvement for NSWindows that don't take long to display. It's particularly appropriate for NSWindows that the user might use once or twice but not display continually. The default is NO.

See also: – **isOneShot**

setReleasedWhenClosed:

– (void)**setReleasedWhenClosed:**(BOOL)*flag*

Sets whether the receiver is merely hidden (NO) or hidden and then released (YES) when it receives a **close** message. The default for NSWindow is YES; the default for NSPanel is NO.

Another strategy for releasing an NSWindow is to have its delegate autorelease it on receiving a **windowShouldClose:** message.

See also: – **close**, – **isReleasedWhenClosed**

setRepresentedFilename:

– (void)**setRepresentedFilename:**(NSString *)*path*

Sets the name of the file that the receiver represents to *path*.

See also: – **representedFilename**, – **setTitleWithRepresentedFilename:**

setResizeIncrements:

– (void)**setResizeIncrements:**(NSSize)*increments*

Restricts the user's ability to resize the window so that the width and height change by multiples of *increments.width* and *increments.height* as the user resizes the window. The width and height increments should be whole numbers, 1.0 or greater. Whatever the current resize increments, you can set an NSWindow's size to any height and width programmatically.

Resize increments and aspect ratio are mutually exclusive attributes. For more information, see **setAspectRatio:**.

See also: – **resizeIncrements**, – **setFrame:display:**

setTitle:

– (void)**setTitle:**(NSString *)*aString*

Sets the string that appears in the receiver's title bar (if it has one) to *aString* and displays the title. Also sets the title of the receiver's miniwindow.

See also: – **title**, – **setTitleWithRepresentedFilename:**, – **setMiniwindowTitle:**

setTitleWithRepresentedFilename:

– (void)**setTitleWithRepresentedFilename:**(NSString *)*path*

Sets *path* as the receiver's title, formatting it as a file system path, and records *path* as the receiver's associated filename using **setRepresentedFilename:**. The title format varies with the platform. On Mach-based systems, the filename is displayed first, followed by an em dash and the path for the directory containing the file. The em dash is offset by two spaces on either side. For example:

MyFile — /Net/server/group/home

This method also sets the title of the receiver's miniwindow.

See also: – **title**, – **setTitle:**, – **setMiniwindowTitle:**

setViewsNeedDisplay:

– (void)**setViewsNeedDisplay:**(BOOL)*flag*

Sets whether the receiver's views need display (YES) or do not need display (NO). You should rarely need to invoke this method; NSView's **setNeedsDisplay:** and similar methods invoke it automatically.

See also: – **viewsNeedDisplay**

stringWithSavedFrame

– (NSString *)**stringWithSavedFrame**

Returns a string that represents the receiver's frame rectangle in a format that can be used with a later **setFrameUsingName:** message.

styleMask

– (unsigned int)**styleMask**

Returns the receiver's style mask, indicating what kinds of control items it displays. See the information about the style mask in the **initWithContentRect:styleMask:backing:defer:** method description. An NSWindow's style is set when the object is initialized. Once set, it can't be changed.

title

– (NSString *)**title**

Returns the string that appears in the title bar of the receiver.

See also: – **setTitle:**, – **setTitleWithRepresentedFilename:**

tryToPerform:with:

– (BOOL)**tryToPerform:**(SEL)*anAction* **with:**(id)*anObject*

Dispatches action messages. The receiver tries to perform the method *anAction* using its inherited NSResponder method **tryToPerform:with:**. If the receiver doesn't perform *anAction*, the delegate is given

the opportunity to perform it using its inherited NSObject method **performSelector:withObject:**. If either the receiver or its delegate accepts *anAction*, this method returns YES; otherwise it returns NO.

unregisterDraggedTypes

– (void)**unregisterDraggedTypes**

Unregisters the receiver as a possible destination for dragging operations.

See also: – **registerForDraggedTypes:**

update

– (void)**update**

The default implementation of this method does nothing more than post an NSWindowDidUpdateNotification to the default notification center. A subclass can override this method to perform specialized operations, but should send an **update** message to **super** just before returning. For example, the NSMenu class implements this method to disable and enable menu commands.

An NSWindow is automatically sent an **update** message on every pass through the event loop and before it's displayed on screen. You can manually cause an update message to be sent to all visible NSWindows through NSApplication's **updateWindows** method.

See also: – **setWindowsNeedUpdate:** (NSApplication), – **updateWindows** (NSApplication)

useOptimizedDrawing:

– (void)**useOptimizedDrawing:**(BOOL)*flag*

Informs the receiver whether to optimize focusing and drawing when displaying its NSViews. The optimizations may prevent sibling subviews from being displayed in the correct order—which matters only if the subviews overlap. You should always set *flag* to YES if there are no overlapping subviews within the NSWindow. The default is NO.

validRequestorForSendType:returnType:

– (id)**validRequestorForSendType:**(NSString *)*sendType* **returnType:**(NSString *)*returnType*

Searches for an object that responds to a Services request by providing input of *sendType* and accepting output of *returnType*. Returns that object, or **nil** if none is found.

Messages to perform this method are initiated by the Services menu. It's part of the mechanism that passes **validRequestorForSendType:returnType:** messages up the responder chain. See the Services documentation in *Programming Topics* for more information.

This method works by forwarding the message to the receiver's delegate if it responds (and provided it isn't an NSResponder with its own next responder). If the delegate doesn't respond to the message or returns **nil** when sent it, this method forwards the message to the NSApplication object. If the NSApplication object returns **nil**, this method also returns **nil**. Otherwise this method returns the object returned by the delegate or the NSApplication object.

See also: – **validRequestorForSendType:returnType:** (NSResponder and NSApplication)

viewsNeedDisplay

– (BOOL)**viewsNeedDisplay**

Returns YES if any of the receiver's NSView's need to be displayed, NO otherwise.

See also: – **setViewsNeedDisplay:**

windowHandle

– (void *)**windowHandle**

Returns a Microsoft Windows HWND handle as a pointer to **void**. This value can be cast directly to HWND. This method exists only on Microsoft Windows; don't attempt to invoke it on Mach or Macintosh.

windowNumber

– (int)**windowNumber**

Returns the window number of the receiver's PostScript window device. Each window device in an application is given a unique window number—note that this isn't the same as the global window number assigned by the Window Server. This number can be used to identify the window device with the **orderWindow:relativeTo:** method and in the Application Kit functions **NSWindowList** and **NSConvertWindowNumberToGlobal**.

If the receiver doesn't have a window device, the value returned will be equal to or less than 0.

See also: – **initWithContentRect:styleMask:backing:defer:**, – **setOneShot:**

worksWhenModal

– (BOOL)**worksWhenModal**

Returns YES if the receiver is able to receive keyboard and mouse events even when some other window is being run modally, NO otherwise. NSWindow's implementation of this method returns NO. Only subclasses of NSPanel should override this default.

See also: – **setWorksWhenModal:** (NSPanel)

zoom:

– (void)**zoom:**(id)*sender*

Toggles the size and location of the window between its standard state (provided by the application as the "best" size to display the window's data) and its user state (a new size and location the user may have set by moving or resizing the window). For more information on the standard and user states, see **windowWillUseStandardFrame:defaultFrame:**.

The **zoom:** method is typically invoked after a user clicks the window's zoom box but may also be invoked programmatically from the **performZoom:** method. It performs the following steps:

1. Invokes the **windowWillUseStandardFrame:defaultFrame:** method, if the delegate or the window class implements it, to obtain a "best fit" frame for the window. If neither the delegate nor the window class implements the method, uses a default frame that nearly fills the current screen, which is defined to be the screen containing the largest part of the window's current frame
2. Adjusts the resulting frame, if necessary, to fit on the current screen.
3. Compares the resulting frame to the current frame to determine whether the window's standard frame is currently displayed. If the current frame is within a few pixels of the standard frame in size and location, it is considered a match.
4. Determines a new frame. If the window is currently in the standard state, the new frame represents the user state, saved during a previous zoom. If the window is currently in the user state, the new frame represents the standard state, computed in step 1. above. If there is no saved user state because there has been no previous zoom, the size and location of the window does not change.
5. Determines whether the window currently allows zooming. By default, zooming is allowed. If the window's delegate implements the **windowShouldZoom:toFrame:** method, **zoom:** invokes that method. If the delegate doesn't implement the method but the window does, **zoom:** invokes the window's version. **windowShouldZoom:toFrame:** returns NO if zooming is not currently allowed.
6. If the window currently allows zooming, sets the new frame.

Methods Implemented By the Delegate

windowDidBecomeKey:

– (void)**windowDidBecomeKey:**(NSNotification *)*aNotification*

Sent by the default notification center immediately after an NSWindow has become key. *aNotification* is always NSWindowDidBecomeKeyNotification. You can retrieve the NSWindow object in question by sending **object** to *aNotification*.

windowDidBecomeMain:

– (void)**windowDidBecomeMain:**(NSNotification *)*aNotification*

Sent by the default notification center immediately after an NSWindow has become main. *aNotification* is always NSWindowDidBecomeMainNotification. You can retrieve the NSWindow object in question by sending **object** to *aNotification*.

windowDidChangeScreen:

– (void)**windowDidChangeScreen:**(NSNotification *)*aNotification*

Sent by the default notification center immediately after an NSWindow has changed screens. *aNotification* is always NSWindowDidChangeScreenNotification. You can retrieve the NSWindow object in question by sending **object** to *aNotification*.

windowDidDeminiaturize:

– (void)**windowDidDeminiaturize:**(NSNotification *)*aNotification*

Sent by the default notification center immediately after an NSWindow has been deminiaturized. *aNotification* is always NSWindowDidDeminiaturizeNotification. You can retrieve the NSWindow object in question by sending **object** to *aNotification*.

windowDidExpose:

– (void)**windowDidExpose:**(NSNotification *)*aNotification*

Sent by the default notification center immediately after an NSWindow has been exposed. *aNotification* is always NSWindowDidExposeNotification. You can retrieve the NSWindow object in question by sending **object** to *aNotification*.

windowDidMiniaturize:

– (void)**windowDidMiniaturize:**(NSNotification *)*aNotification*

Sent by the default notification center immediately after an NSWindow has been miniaturized. *aNotification* is always NSWindowDidMiniaturizeNotification. You can retrieve the NSWindow object in question by sending **object** to *aNotification*.

windowDidMove:

– (void)**windowDidMove:**(NSNotification *)*aNotification*

Sent by the default notification center immediately after an NSWindow has been moved. *aNotification* is always NSWindowDidMoveNotification. You can retrieve the NSWindow object in question by sending **object** to *aNotification*.

windowDidResignKey:

– (void)**windowDidResignKey:**(NSNotification *)*aNotification*

Sent by the default notification center immediately after an NSWindow has resigned its status as key window. *aNotification* is always NSWindowDidResignKeyNotification. You can retrieve the NSWindow object in question by sending **object** to *aNotification*.

windowDidResignMain:

– (void)**windowDidResignMain:**(NSNotification *)*aNotification*

Sent by the default notification center immediately after an NSWindow has resigned its status as main window. *aNotification* is always NSWindowDidResignMainNotification. You can retrieve the NSWindow object in question by sending **object** to *aNotification*.

windowDidResize:

– (void)**windowDidResize:**(NSNotification *)*aNotification*

Sent by the default notification center immediately after an NSWindow has been resized. *aNotification* is always NSWindowDidResizeNotification. You can retrieve the NSWindow object in question by sending **object** to *aNotification*.

windowDidUpdate:

– (void)**windowDidUpdate:**(NSNotification *)*aNotification*

Sent by the default notification center immediately after an NSWindow receives an **update** message. *aNotification* is always NSWindowDidUpdateNotification. You can retrieve the NSWindow object in question by sending **object** to *aNotification*.

windowShouldClose:

– (BOOL)**windowShouldClose:**(id)*sender*

Invoked when the user attempts to close the window or when the NSWindow receives a **performClose:** message. The delegate can return NO to prevent *sender* from closing.

windowShouldZoom:toFrame:

– (BOOL)**windowShouldZoom:**(NSWindow *)*sender* **toFrame:**(NSRect)*newFrame*

Invoked just before *sender* is zoomed. Zooming will change the frame of *sender* to *newFrame*. The delegate can return NO to prevent *sender* from zooming.

See also: – **windowWillUseStandardFrame:defaultFrame:**

windowWillClose:

– (void)**windowWillClose:**(NSNotification *)*aNotification*

Sent by the default notification center immediately before an NSWindow closes. *aNotification* is always NSWindowWillCloseNotification. You can retrieve the NSWindow object in question by sending **object** to *aNotification*.

windowWillMiniaturize:

– (void)**windowWillMiniaturize:**(NSNotification *)*aNotification*

Sent by the default notification center immediately before an NSWindow is miniaturized. *aNotification* is always NSWindowWillMiniaturizeNotification. You can retrieve the NSWindow object in question by sending **object** to *aNotification*.

windowWillMove:

– (void)**windowWillMove:**(NSNotification *)*aNotification*

Sent by the default notification center immediately before an NSWindow is moved. *aNotification* is always NSWindowWillMoveNotification. You can retrieve the NSWindow object in question by sending **object** to *aNotification*.

windowWillResize:toSize:

– (NSSize)**windowWillResize:**(NSWindow *)*sender* **toSize:**(NSSize)*proposedFrameSize*

Invoked when *sender* is being resized (whether by the user or through one of the **setFrame...** methods other than **setFrame:display:**). *proposedFrameSize* contains the size (in screen coordinates) that the *sender* will be resized to. To resize to a different size, simply return the desired size from this method; to avoid resizing, return the current size. The NSWindow's minimum and maximum size constraints have already been applied when this method is invoked.

While the user is resizing an NSWindow, the delegate is sent a series of **windowWillResize:toSize:** messages as the NSWindow's outline is dragged. The NSWindow's outline is displayed at the constrained size as set by this method.

windowWillReturnFieldEditor:toObject:

– (id)**windowWillReturnFieldEditor:**(NSWindow *)*sender* **toObject:**(id)*anObject*

Invoked when the field editor of *sender* is requested by *anObject*. If the delegate's implementation of this method returns an object other than **nil**, the NSWindow substitutes it for the field editor and returns it to *anObject*.

See also: – **fieldEditor:forObject:**

windowWillUseStandardFrame:defaultFrame:

– (NSRect)**windowWillUseStandardFrame:**(NSWindow *)*sender*
defaultFrame (NSRect)*defaultFrame*

Invoked by the **zoom:** method while determining a frame the window may be zoomed to. Returns the standard frame (described below) for *window*. The *defaultFrame* parameter passed in is the size of the current screen, which is the screen containing the largest part of the window's current frame, possibly reduced on the top, bottom, left, or right, depending on the current interface style. For the Macintosh style, for example, the frame is reduced on the top to leave room for the menu bar.

The standard frame for a window should supply the size and location that are "best" for the type of information shown in the window, taking into account the available display or displays. For example, the

best width for a window that displays a word-processing document is the width of a page or the width of the display, whichever is smaller. The best height can be determined similarly. On return from this method, the **zoom:** method modifies the returned standard frame, if necessary, to fit on the current screen.

To customize the standard state, you implement **windowWillUseStandardFrame:defaultFrame:** in the class of the window's delegate or, if necessary, in a window subclass. Your version should return a suitable standard frame, based on the currently displayed data or other factors.

See also: – **windowShouldZoom:toFrame:**, – **zoom:** (NSWindow class)

Notifications

NSWindowDidBecomeKeyNotification

Posted whenever the NSWindow becomes the key window.

This notification contains a notification object but no userInfo dictionary. The notification object is the NSWindow that has become key.

NSWindowDidBecomeMainNotification

Posted whenever the NSWindow becomes the main window.

This notification contains a notification object but no userInfo dictionary. The notification object is the NSWindow that has become main.

NSWindowDidChangeScreenNotification

Posted whenever a portion of the NSWindow's frame moves onto or off of a screen.

This notification contains a notification object but no userInfo dictionary. The notification object is the NSWindow that has changed screens.

NSWindowDidDeminiaturizeNotification

Posted whenever the NSWindow is deminiaturized.

This notification contains a notification object but no userInfo dictionary. The notification object is the NSWindow that has been deminiaturized.

NSWindowDidExposeNotification

Posted whenever a portion of a nonretained NSWindow is exposed, whether by being ordered in front of other windows or by other widows being removed from in front of it.

This notification contains a notification object and a userInfo dictionary. The notification object is the NSWindow that has been exposed. The userInfo dictionary contains the key NSExposedRect and an associated value for the rectangle that has been exposed.

NSWindowDidMiniaturizeNotification

Posted whenever the NSWindow is miniaturized.

This notification contains a notification object but no userInfo dictionary. The notification object is the NSWindow that has been miniaturized.

NSWindowDidMoveNotification

Posted whenever the NSWindow is moved.

This notification contains a notification object but no userInfo dictionary. The notification object is the NSWindow that has moved.

NSWindowDidResignKeyNotification

Posted whenever the NSWindow resigns its status as key window.

This notification contains a notification object but no userInfo dictionary. The notification object is the NSWindow that has resigned its key window status.

NSWindowDidResignMainNotification

Posted whenever the NSWindow resigns its status as main window.

This notification contains a notification object but no userInfo dictionary. The notification object is the NSWindow that has resigned its main window status.

NSWindowDidResizeNotification

Posted whenever the NSWindow’s size changes.

This notification contains a notification object but no userInfo dictionary. The notification object is the NSWindow whose size has changed.

NSWindowDidUpdateNotification

Posted whenever the NSWindow receives an **update** message.

This notification contains a notification object but no userInfo dictionary. The notification object is the NSWindow that received the **update** message.

NSWindowWillCloseNotification

Posted whenever the NSWindow is about to close.

This notification contains a notification object but no userInfo dictionary. The notification object is the NSWindow that is about to close.

NSWindowWillMiniaturizeNotification

Posted whenever the NSWindow is about to be miniaturized.

This notification contains a notification object but no userInfo dictionary. The notification object is the NSWindow that is about to be miniaturized.

NSWindowWillMoveNotification

Posted whenever the NSWindow is about to move.

This notification contains a notification object but no userInfo dictionary. The notification object is the NSWindow that is about to move.

NSWindowController

Inherits From: NSObject

Conforms To: NSObject (NSObject)
NSCoding

Declared In: AppKit/NSWindowController.h

Class Description

An NSWindowController object manages a window, usually a window stored in a nib file. This management entails:

- Loading and displaying the window
- Closing the window when appropriate
- Customizing the window's title
- Storing the window's frame (size and location) in the defaults database
- Cascading the window in relation to other document windows of the application

An NSWindowController can manage a window by itself or as a role player in the Application Kit's document-based architecture, which also includes NSDocument and NSDocumentController objects. In this architeture, an NSWindowController is created and managed by a "document" (an instance of an NSDocument subclass) and, in turn, keeps a reference to the document. For a discussion of this architecture, see ""Document-Based Application Architecture"" in the NSDocument specification..

The relationship between an NSWindowController (or, simply, a window controller) and a nib file is important. Although a window controller can manage a programmatically created window, it usually manages a window in a nib file. The nib file can contain other top-level objects, including other windows, but the window controller's responsibility is this primary window. The window controller is usually the owner of the nib file, even when it is part of a document-based application. Regardless of who is the file's owner, the window controller is responsible for freeing all top-level objects in the nib file it loads.

For simple documents—that is, documents with only one nib file containing a window—you need do little directly with NSWindowController. The Application Kit will create one for you. However, if the default window controller is not sufficient, you can create a custom subclass of NSWindowController. For documents with multiple windows or panels, your document must create separate instances of NSWindowController (or of custom subclasses of NSWindowController), one for each window or panel. An example is a CAD application that has different windows for side, top, and front views of drawn objects. What you do in your NSDocument subclass determines whether the default NSWindowController or separately created and configured NSWindowController objects are used.

Window Closing Behavior

When a window is closed and it is part of a document-based application, the document removes the window's NSWindowController from its list of window controllers. This results in the deallocation of the window controller and the window, and possibly the deallocation of the NSDocument object itself. But when a window controller is not part of a document-based application, closing the window by default does not result in the deallocation of the window or window controller. This is desired behavior for a window controller that manages something like an Inspector panel; you shouldn't have to load the nib file again and recreate the objects the next time the user requests the Inspector.

If you want the closing of a window to make both window and window controller go away when it isn't part of a document, your subclass of NSWindowController can observe NSWindow's "NSWindowWillCloseNotification" or, as window delegate, implement the **windowWillClose:** method and in your implementations include the following line of code:

```
[self autorelease];
```

As a consequence of autoreleasing itself, a NSWindowController object autoreleases its window as well as all other top-level objects in its nib file. In any case, a window controller should not get rid of its window until it ensures its own deallocation.

Subclassing NSWindowController

You should create a subclass of NSWindowController when you want to augment the default behavior, such as to give the window a custom title or to perform some setup tasks before the window is loaded. In your class's initialization method, be sure to invoke on **super** either one of the **initWithWindowNibName:...** initializers or the **initWithWindow:** initializer. Which one depends on whether the window object originates in a nib file or is programmatically created.

Three NSWindowController methods are most commonly overriden:

- **windowWillLoad**. Override to perform tasks before the window nib file is loaded.
- **windowDidLoad**. Override to perform tasks after the window nib file is loaded.
- **windowTitleForDisplayName:**. Override to customize the window title.

You can also override **loadWindow** to get different nib-searching or nib-loading behavior, although there is usually no need to do this.

Adopted Protocols

NSCoding

– encodeWithCoder:
– initWithCoder:

Method Types

Initializing NSWindowControllers

– initWithWindow:
– initWithWindowNibName:
– initWithWindowNibName:owner:

Loading and display the window

– loadWindow
– showWindow:
– isWindowLoaded
– window
– windowDidLoad
– windowWillLoad

Setting and getting the document

– setDocument:
– document

Closing the window

– close
– shouldCloseDocument
– setShouldCloseDocument:

Getting nib file information

– owner
– windowNibName

Setting and getting window attributes

– setShouldCascadeWindows:
– shouldCascadeWindows
– setWindowFrameAutosaveName:
– windowFrameAutosaveName
– windowTitleForDocumentDisplayName:

Instance Methods

close

– (void)**close**

Closes the window if it was loaded. Because this method closes the window without asking the user for confirmation, you usually do not invoke it when the Close menu command is chosen. Instead invoke NSWindow's **performClose:** on the receiver's window. See "“Window Closing Behavior”" in the class description for an overview of deallocation behavior when a window is closed.

See also: – **shouldCloseDocument**, – **setShouldCloseDocument:**

document

– (id)**document**

Returns the NSDocument object associated with the receiver or **nil** if there is none. If the receiver is part of a document-based application, it typically keeps a reference to its NSDocument. When a window is loaded or the document changes, the NSWindowController gets the edited status from the document and sets the window with it (NSWindow's **setDocumentEdited:**). The Application Kit also uses this outlet to access the document for relevant next-responder messages.

See also: – **setDocument:**

initWithWindow:

– (id)**initWithWindow:**(NSWindow *)*window*

Returns an NSWindowController object initialized with *window*, the window object to manage. The *window* argument can be **nil**.This initializer is useful when a window has been loaded but no window controller is assigned. This is the designated initializer for NSWindowController. The default initialization turns on cascading, sets the **shouldCloseDocument** flag to NO, and sets the window frame autosave name to an empty string. As a side effect, the created window controller is added as an observer of the NSWindowWillCloseNotifications posted by that window object (which is handled by a private method). If you make the window controller a delegate of the window, you can implement NSWindow's **windowShouldClose:** delegate method.

initWithWindowNibName:

– (id)**initWithWindowNibName:**(NSString *)*windowNibName*

Returns an NSWindowController object initialzed with *windowNibName*, the name of the nib file (minus the ".nib" extension) that archives the receiver's window. The *windowNibName* argument cannot be **nil**. Sets the owner of the nib file to the receiver. The default initialization turns on cascading, sets the **shouldCloseDocument** flag to NO, and sets the autosave name for the window's frame to an empty string.

initWithWindowNibName:owner:

– (id)**initWithWindowNibName:**(NSString *)*windowNibName* **owner:**(id)*owner*

Returns an NSWindowController object initialzed with *windowNibName* and *owner*. Neither *windowNibName* or *owner* can be **nil**. The *windowNibName* argument is the name of the nib file (minus the ".nib" extension) that archives the receiver's window. The *owner* argument is the nib file's owner. The default initialization turns on cascading, sets the **shouldCloseDocument** flag to NO, and sets the autosave name for the window's frame to an empty string.

isWindowLoaded

– (BOOL)**isWindowLoaded**

Returns whether the nib file containing the receiver's window has been loaded.

See also: **– loadWindow**, **– window**, **– windowDidLoad**, **– windowWillLoad**

loadWindow

– (void)**loadWindow**

Loads the receiver's window from the nib file. You should never directly invoke this method. Instead, invoke **window** so that the **windowDidLoad** and **windowWillLoad** methods are invoked. Subclasses can override this method if the way it finds and loads the window is not adequate. It uses NSBundle's **bundleForClass:** method to get the bundle, using the class of the nib file owner as argument. It then locates the nib file within the bundle and, if successful, loads it; if unsuccessful, it tries to find the nib file in the main bundle.

See also: **– isWindowLoaded**

owner

– (id)**owner**

Returns the owner of the nib file containing the window managed by the receiver. This is usually **self**, but can be the receiver's document (an instance of an NSDocument subclass) or some other object.

See also: **– windowNibName**

setDocument:

– (void)**setDocument:**(NSDocument *)*document*

Sets the document associated with the window managed by the receiver. A document is an instance of an NSDocument subclass that represents and manages the data displayed and captured in the window. A window controller has a reference to its document document when it is part of a document-based application.

See also: **– document**

setShouldCascadeWindows:

– (void)**setShouldCascadeWindows:**(BOOL)*flag*

Sets whether the window, when it is displayed, should cascade in relation to other document windows (that is, have a slightly offset location so that the title bars of previously displayed windows are still visible). The default is YES.

See also: – **shouldCascadeWindows**

setShouldCloseDocument:

– (void)**setShouldCloseDocument:**(BOOL)*flag*

Sets whether the receiver should close the associated document when the window it manages is closed (*flag* is YES) or whether to close the document only when the last document window has been closed (*flag* is NO). The default is NO.

See also: – **shouldCloseDocument**

setWindowFrameAutosaveName:

– (void)**setWindowFrameAutosaveName:**(NSString *)*name*

Sets the name under which the window's frame (it's size and location on the screen) is saved in the defaults database. By default, *name* is an empty string, causing no information to be stored in the defaults database.

See also: – **windowFrameAutosaveName**, **– setFrameAutosaveName:** (NSWindow)

shouldCascadeWindows

– (BOOL)**shouldCascadeWindows**

Returns whether the window will cascade in relation to other document windows when it is displayed.

See also: – **setShouldCascadeWindows:**

shouldCloseDocument

– (BOOL)**shouldCloseDocument**

Returns whether the receiver closes the associated document when the window it manages is closed (YES) or whether the document is closed only when last remaining window of the document is closed (NO).

See also: – **setShouldCloseDocument:**

showWindow:

– (void)**showWindow:**(id)*sender*

Displays the window associated with the receiver. If the window is an NSPanel object and has its **becomesKeyOnlyIfNeeded** flag set to YES, the window is displayed in front of all other windows but is not made key; otherwise it is displayed in front and is made key. This method is useful for menu actions.

See also: – **makeKeyAndOrderFront:** (NSWindow), – **orderFront:** (NSWindow)

window

– (NSWindow *)**window**

Returns the window owned by the receiver or **nil** if there isn't one. If the window has not yet been loaded, it attempts to load the window's nib file using **loadWindow**. Before it loads the window, it invokes **windowWillLoad** in subclass implementations and, if the NSWindowController has a document, it invokes the NSDocument's corresponding method **windowControllerWillLoadNib:** (if implemented). After loading the window, it invokes **windowDidLoad** and, if there is a document, the NSDocument method **windowControllerDidLoadNib:** (if implemented).

See also: – **windowControllerWillLoadWindowNib:** (NSDocument)

windowDidLoad

– (void)**windowDidLoad**

Allows subclasses of NSWindowController to perform any required tasks after the window owned by the receiver has been loaded. The default implementation does nothing.

See also: – **loadWindow**, – **window**, – **windowWillLoad**

windowFrameAutosaveName

– (NSString *)**windowFrameAutosaveName**

Returns the name under which the frame rectangle of the window owned by the receiver is stored in the defaults database.

See also: – **setWindowFrameAutosaveName:**

windowNibName

– (NSString *)**windowNibName**

Returns the name of the nib file that stores the window associated with the receiver. The name does not include the ".nib" extension.

See also: – **owner**

windowTitleForDocumentDisplayName:

– (NSString *)**windowTitleForDocumentDisplayName:**(NSString *)*displayName*

Returns *displayName* by default. The display name, which is generally maintained by the associated NSDocument, is the last path component under which the document file is saved. Subclasses can override this method to customize the window title. For example, a CAD application could append "-Top" or "-Side", depending on the view displayed by the window.

windowWillLoad

– (void)**windowWillLoad**

Allows subclasses of NSWindowController to perform any required tasks before the window owned by the receiver is loaded. The default implementation does nothing.

See also: – **loadWindow**, – **window**, – **windowDidLoad**

NSWorkspace

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: AppKit/NSWorkspace.h

Class Description

An NSWorkspace object responds to application requests to perform a variety of services:

- opening, manipulating, and obtaining information about files and devices
- tracking changes to the file system, devices, and the user database
- launching applications
- miscellaneous services such as animating an image and requesting additional time before power off

There is one shared NSWorkspace object per application. You use the class method **sharedWorkspace** to access it. For example, the following statement uses an NSWorkspace object to request that a file be opened in the Edit application:

```
[[NSWorkspace sharedWorkspace] openFile:@"/Myfiles/README"
    withApplication:@"Edit"];
```

Note: On the Microsoft Windows platform, some of the methods in this class have no effect. Refer to the method descriptions below.

Method Types

Accessing the shared NSWorkspace

+ sharedWorkspace

Accessing the NSWorkspace notification center

– notificationCenter

Opening files

– openFile:
– openFile:withApplication:
– openFile:fromImage:at:inView:
– openFile:withApplication:andDeactivate:
– openTempFile:

Manipulating applications

– launchApplication:
– launchApplication:showIcon:autolaunch:
– hideOtherApplications

Manipulating files

– performFileOperation:source:destination:files:tag:
– selectFile:inFileViewerRootedAtPath:

Requesting information about files

– iconForFile:
– iconForFileType:
– iconForFiles:
– getInfoForFile:application:type:
– fullPathForApplication:
– getFileSystemInfoForPath:isRemovable:isWritable:
isUnmountable:description:type:

Requesting additional time before logout

– extendPowerOffBy:

Tracking changes to the file system

– noteFileSystemChanged
– fileSystemChanged

Updating registered services and file types

– findApplications

Tracking changes to the defaults database

– noteUserDefaultsChanged
– userDefaultsChanged

Tracking status changes for applications and devices

– mountedRemovableMedia
– mountNewRemovableMedia
– checkForRemovableMedia

Animating an image

– slideImage:from:to:

Unmounting a device

– unmountAndEjectDeviceAtPath:

Class Methods

sharedWorkspace

+ (NSWorkspace *)**sharedWorkspace**

Returns the shared NSWorkspace instance.

Instance Methods

checkForRemovableMedia

– (void)**checkForRemovableMedia**

On the Mach platform, polls the system's drives for any disks that have been inserted but not yet mounted. **checkForRemovableMedia** doesn't wait until such disks are mounted; instead, it requests that the disk be mounted asynchronously and returns immediately.

This method has no effect on the Microsoft Windows platform.

See also: – **mountNewRemovableMedia**, – **mountedRemovableMedia**

extendPowerOffBy:

– (int)**extendPowerOffBy:**(int)*requested*

Requests *requested* milliseconds more time before the power goes off or the user logs out. Returns the number of additional milliseconds granted. On some platforms you might not be able to extend the time.

fileSystemChanged

– (BOOL)**fileSystemChanged**

On the Mach platform, returns YES if a change to the file system has been registered with a **noteFileSystemChanged** message since the last **fileSystemChanged** message; NO otherwise.

This method is not implemented on the Microsoft Windows platform. If you try to use it, it raises an NSInvalidArgumentException.

findApplications

– (void)**findApplications**

On the Mach platform, examines all applications in the normal places (**/Network/Applications**, **/System/Applications**, **/System/Developer/Applications**) and updates the records of registered services and file types.

This method has no effect on the Microsoft Windows platform.

fullPathForApplication:

– (NSString *)**fullPathForApplication:**(NSString *)*appName*

Returns the full path for the application *appName*, or **nil** if *appName* isn't in one of the normal places.

getFileSystemInfoForPath:isRemovable:isWritable:isUnmountable:description:type:

– (BOOL)**getFileSystemInfoForPath:**(NSString *)*fullPath*
isRemovable:(BOOL *)*removableFlag*
isWritable:(BOOL *)*writableFlag*
isUnmountable:(BOOL *)*unmountableFlag*
description:(NSString **)*description*
type:(NSString **)*fileSystemType*

On the Mach platform, describes the file system at *fullPath*. This method has no effect on the Microsoft Windows platform.

Returns YES if *fullPath* is a file system mount point, NO otherwise. If the return value is YES, *description* describes the file system; this value can be used in strings, but it shouldn't be depended upon by program logic. Example values for description are "hard," "nfs," and "foreign." *fileSystemType* indicates the file system type; values could be "NeXT," "DOS," or other values. *removableFlag* is YES if the file system is on removable media, NO otherwise. *writableFlag* is YES if the file system's media is writable, NO otherwise. *unmountableFlag* returns YES if the file system is unmountable, NO otherwise.

getInfoForFile:application:type:

– (BOOL)**getInfoForFile:**(NSString *)*fullPath*
application:(NSString **)*appName*
type:(NSString **)*type*

Retrieves information about the file specified by *fullPath*. If this method returns YES, the NSString pointed to by *appName* is set to the application the system would use to open *fullPath*. The NSString pointed to by *type* contains one of the following values or a file name extension such as "rtf" indicating the file's type:

Value	Type of File
NSPlainFileType	Plain (untyped) file
NSDirectoryFileType	Directory
NSApplicationFileType	OpenStep application
NSFilesystemFileType	File system mount point
NSShellCommandFileType	Executable shell command

This method returns NO if it could not find *fullPath*.

See also: – **iconForFile:**, – **iconForFiles:**

hideOtherApplications

– (void)**hideOtherApplications**

Hides all applications other than the sender. This method has no effect on the Microsoft Windows platform. On the Mach platform, the user can hide all applications except the current one by Command-double-clicking an application's tile, so programmatic invocation of this method is usually unnecessary.

iconForFile:

– (NSImage *)**iconForFile:**(NSString *)*fullPath*

Returns an NSImage with the icon for the single file specified by *fullPath*.

See also: – **getInfoForFile:application:type:**, – **iconForFileType:**, – **iconForFiles:**

iconForFileType:

– (NSImage *)**iconForFileType:**(NSString *)*fileType*

Returns an NSImage with the icon for the file type specified by *fileType*.

See also: – **iconForFile:**, – **iconForFiles:**

iconForFiles:

– (NSImage *)**iconForFiles:**(NSArray *)*fullPaths*

Returns an NSImage with the icon for the files specified in *fullPaths*, an array of NSStrings. If *fullPaths* specifies one file, its icon is returned. If *fullPaths* specifies more than one file, an icon representing the multiple selection is returned.

See also: – **iconForFile:**, – **iconForFileType:**

launchApplication:

– (BOOL)**launchApplication:**(NSString *)*appName*

Launches the application *appName*. *appName* need not be specified with a full path and, in the case of an application wrapper, can be specified with or without the **.app** extension. Returns YES if the application is successfully launched or already running, NO if it can't be launched.

Before this method begins, it posts an NSWorkspaceWillLaunchApplicationNotification to the NSWorkspace's notification center. When the operation is complete, it posts an NSWorkspaceDidLaunchApplicationNotification.

See also: – **launchApplication:showIcon:autolaunch:**

launchApplication:showIcon:autolaunch:

– (BOOL)**launchApplication:**(NSString *)*appName*
showIcon:(BOOL)*showIcon*
autolaunch:(BOOL)*autolaunch*

Launches the application *appName*. If *showIcon* is NO, the application's icon won't be placed on the screen. (The icon still exists, though.) If *autolaunch* is YES, the autolaunch default will be set as though the application were autolaunched at startup. This method is provided to enable daemon-like applications that lack a normal user interface and for use by alternative dock programs. Its use is not generally encouraged.

Returns YES if the application is successfully launched or already running, and NO if it can't be launched.

Before this method begins, it posts an NSWorkspaceWillLaunchApplicationNotification to the NSWorkspace's notification center. When the operation is complete, it posts an NSWorkspaceDidLaunchApplicationNotification.

See also: – **launchApplication:**

mountNewRemovableMedia

– (NSArray *)**mountNewRemovableMedia**

On the Mach platform, polls the system's drives for any disks that have been inserted but not yet mounted, waits until the new disks have been mounted, and returns an NSArray of NSStrings containing full pathnames to all newly mounted disks. This method posts an NSWorkspaceDidMountNotification to the NSWorkspace's notification center when it is finished.

This method is not implemented on the Microsoft Windows platform. If you try to use it, it raises an NSInvalidArgumentException.

See also: – **checkForRemovableMedia**, – **mountedRemovableMedia**

mountedRemovableMedia

– (NSArray *)**mountedRemovableMedia**

Returns an NSArray of NSStrings containing the full pathnames of all currently mounted removable disks. This method is not implemented on the Microsoft Windows platform. If you try to use it, it raises an NSInvalidArgumentException.

On the Mach platform, if the computer provides an interrupt or other notification when the user inserts a disk into a drive, the Workspace Manager will mount the disk immediately. However, if no notification is given, the Workspace Manager won't be aware that a disk needs to be mounted. On such systems, an application should invoke either **mountNewRemovableMedia** or **checkForRemovableMedia** before invoking **mountedRemovableMedia**. Either of these methods cause the Workspace Manager to poll the drives to see if a disk is present. If a disk has been inserted but not yet mounted, these methods will cause the Workspace Manager to mount it.

The Disk button in an Open or Save panel invokes **mountedRemovableMedia** and **mountNewRemovableMedia** as part of its operation, so most applications won't need to invoke these methods directly.

See also: – **checkForRemovableMedia**, – **mountNewRemovableMedia**

noteFileSystemChanged

– (void)**noteFileSystemChanged**

On the Mach platform, informs NSWorkspace that the file system has changed. NSWorkspace then gets the status of all the files and directories it is interested in and updates itself appropriately. This method is used by many objects that write or delete files.

This method has no effect on the Microsoft Windows platform.

See also: – **fileSystemChanged**

noteUserDefaultsChanged

– (void)**noteUserDefaultsChanged**

On the Mach platform, informs NSWorkspace that the defaults database has changed. NSWorkspace then reads all the defaults it is interested in and reconfigures itself appropriately. For example, on Mach platforms, this method is used by the Preferences application to notify Workspace Manager whether the user prefers to see hidden files.

This method has no effect on the Microsoft Windows platform.

See also: – **userDefaultsChanged**

notificationCenter

– (NSNotificationCenter *)**notificationCenter**

Returns the notification center for workspace notifications.

openFile:

– (BOOL)**openFile:**(NSString *)*fullPath*

Opens the file specified by *fullPath* using the default application for its type; returns YES if file was successfully opened, NO otherwise. The sending application is deactivated before the request is sent.

See also: – **openFile:fromImage:at:inView:**, – **openFile:withApplication:**, – **openFile:withApplication: andDeactivate:**, – **openTempFile:**

openFile:fromImage:at:inView:

– (BOOL)**openFile:**(NSString *)*fullPath*
fromImage:(NSImage *)*anImage*
at:(NSPoint)*point*
inView:(NSView *)*aView*

Opens the file specified by *fullPath* using the default application for its type. On the Mach platform, Workspace Manager provides animation before opening the file to give the user feedback that the file is to be opened. To provide this animation, *anImage* should contain an icon for the file, and its image should be displayed at *point*, specified in *aView*'s coordinates. On the Microsoft Windows platform, this method is the same as the **openFile:** method.

The sending application is deactivated before the request is sent. Returns YES if the file is successfully opened, NO otherwise.

See also: – **openFile:**, – **openFile:withApplication:**, – **openFile:withApplication:andDeactivate:**, – **openTempFile:**

openFile:withApplication:

– (BOOL)**openFile:**(NSString *)*fullPath* **withApplication:**(NSString *)*appName*

Opens the file specified by *fullPath* using the *appName* application. *appName* need not be specified with a full path and, in the case of an application wrapper, can be specified with or without the **.app** extension. The sending application is deactivated before the request is sent. Returns YES if the file is successfully opened, NO otherwise.

See also: – **openFile:**, – **openFile:withApplication:andDeactivate:**

openFile:withApplication:andDeactivate:

– (BOOL)**openFile:**(NSString *)*fullPath*
withApplication:(NSString *)*appName*
andDeactivate:(BOOL)*flag*

Opens the file specified by *fullPath* using the *appName* application. *appName* need not be specified with a full path and, in the case of an application wrapper, can be specified with or without the **.app** extension. If *appName* is **nil**, the default application for the file's type is used. If *flag* is YES, the sending application is deactivated before the request is sent, allowing the opening application to become the active application. Returns YES if the file is successfully opened, NO otherwise.

See also: – **openFile:**, – **openFile:withApplication:**, – **application:openFile:** (NSApplication delegate method)

openTempFile:

– (BOOL)**openTempFile:**(NSString *)*fullPath*

Opens the temporary file specified by *fullPath* using the default application for its type. The sending application is deactivated before the request is sent. Using this method instead of one of the **openFile:...** methods lets the receiving application know that it should delete the file when it no longer needs it. Returns YES if the file is successfully opened, NO otherwise.

See also: – **openFile:**, – **openFile:fromImage:at:inView:**, – **openFile:withApplication:**, – **openFile: withApplication:andDeactivate:**

performFileOperation:source:destination:files:tag:

– (BOOL)**performFileOperation:**(NSString *)*operation*
source:(NSString *)*source*
destination:(NSString *)*destination*
files:(NSArray *)*files*
tag:(int *)*tag*

Performs a file operation on a set of files in a particular directory. *operation* is some file operation, such as compressing or moving files. *files* contains NSString specifying the names of the files to be manipulated. The file names are given relative to the *source* directory. The list can contain both files and directories; all of them must be located directly within source (not in one of its subdirectories).

Some operations—such as moving, copying, and linking files—require a destination directory to be specified. If not, *destination* should be the empty string (@"").

The possible values for *operation* are:

Operation	Meaning
NSWorkspaceMoveOperation	Move file to destination
NSWorkspaceCopyOperation	Copy file to destination
NSWorkspaceLinkOperation	Create link to file in destination
NSWorkspaceCompressOperation	Compress file
NSWorkspaceDecompressOperation	Decompress file
NSWorkspaceEncryptOperation	Encrypt file
NSWorkspaceDecryptOperation	Decrypt file
NSWorkspaceDestroyOperation	Destroy file

Operation	Meaning
NSWorkspaceRecycleOperation	Move file to recycler
NSWorkspaceDuplicateOperation	Duplicate file in source directory

Note: NSWorkspaceCompressOperation, NSWorkspaceDecompressOperation, NSWorkspaceEncryptOperation, and NSWorkspaceDecryptOperation are not available on the Microsoft Windows platform.

This method returns YES if the operation succeeded, NO otherwise, In *tag*, the method returns a negative integer if the operation fails, 0 if the operation is performed synchronously and succeeds, and a positive integer if the operation is performed asynchronously. The positive integer is a tag that identifies the requested file operation. Before this method returns, it posts an NSWorkspaceDidPerformFileOperationNotification to NSWorkspace's notification center.

selectFile:inFileViewerRootedAtPath:

– (BOOL)**selectFile:**(NSString *)*fullPath* **inFileViewerRootedAtPath:**(NSString *)*rootFullPath*

Selects the file specified by *fullPath*. If a path is specified by *rootFullPath*, a new file viewer is opened. If *rootFullPath* is an empty string (@""), the file is selected in the main viewer. Returns YES if the file is successfully selected, NO otherwise.

slideImage:from:to:

– (void)**slideImage:**(NSImage *)*image*
from:(NSPoint)*fromPoint*
to:(NSPoint)*toPoint*

On Mach platforms, animates a sliding image of *image* from *fromPoint* to *toPoint*, specified in screen coordinates. This method has no effect on the Microsoft Windows platform.

unmountAndEjectDeviceAtPath:

– (BOOL)**unmountAndEjectDeviceAtPath:**(NSString *)*path*

Unmounts and ejects the device at path. Returns YES if unmount succeeded and NO otherwise. When this method begins, it posts an NSWorkspaceWillUnmountNotification to NSWorkspace's notification center. When it is finished, it posts an NSWorkspaceDidUnmountNotification.

userDefaultsChanged

– (BOOL)**userDefaultsChanged**

On the Mach platform, returns whether a change to the defaults database has been registered with a **noteUserDefaultsChanged** message since the last **userDefaultsChanged** message.

This method has no effect on the Microsoft Windows platform.

Notifications

All NSWorkspace notifications are posted to NSWorkspace's own notification center, not the application's default notification center. Access this center using the **notificationCenter** method.

```
NSNotificationCenter *workspaceCenter = [[NSWorkspace sharedWorkspace]
                                 notificationCenter];
```

NSWorkspaceDidLaunchApplicationNotification

Posted when a new application has started up.

This notification contains a notification object and a userInfo dictionary. The notification object is the shared NSWorkspace instance. The userInfo dictionary contains these keys and values:

Key	Value
NSApplicationName	The application being terminated

NSWorkspaceDidMountNotification

Posted when a new device has been mounted.

This notification contains a notification object and a userInfo dictionary. The notification object is the shared NSWorkspace instance. The userInfo dictionary contains these keys and values:

Key	Value
NSDevicePath	The path where the device was mounted

NSWorkspaceDidPerformFileOperationNotification

Posted when a file operation has been performed.

This notification contains a notification object and a userInfo dictionary. The notification object is the shared NSWorkspace instance. The userInfo dictionary contains these keys and values:

Key	Value
NSOperationNumber	A number indicating the type of file operation completed

NSWorkspaceDidTerminateApplicationNotification

Posted when an application finishes executing.

This notification contains a notification object and a userInfo dictionary. The notification object is the shared NSWorkspace instance. The userInfo dictionary contains these keys and values:

Key	Value
NSApplicationName	The application that terminated

NSWorkspaceDidUnmountNotification

Posted when the workspace has unmounted a device.

This notification contains a notification object and a userInfo dictionary. The notification object is the shared NSWorkspace instance. The userInfo dictionary contains these keys and values:

Key	Value
NSDevicePath	The path where the device was previously mounted

NSWorkspaceWillLaunchApplicationNotification

Posted when the workspace is about to launch an application.

This notification contains a notification object and a userInfo dictionary. The notification object is the shared NSWorkspace instance. The userInfo dictionary contains these keys and values:

Key	Value
NSApplicationName	The application about to be launched

NSWorkspaceWillPowerOffNotification

Posted when the user has requested that the machine be powered off.

This notification contains a notification object but no userInfo dictionary. The notification object is the shared NSWorkspace instance.

NSWorkspaceWillUnmountNotification

Posted when the workspace is about to unmount a device.

This notification contains a notification object and a userInfo dictionary. The notification object is the shared NSWorkspace instance. The userInfo dictionary contains these keys and values:

Key	Value
NSDevicePath	The path where the device is mounted

Functions

This section describes functions and function-like macros available in the Application Kit library.

NSApplicationMain

SUMMARY This function is called by the **main** function to create and run the application.

DECLARED IN AppKit/NSApplication.h

SYNOPSIS int **NSApplicationMain(**int argc, const char *argv[]**)**

DESCRIPTION The **NSApplicationMain** function creates the application, loads the main nib file from the application's main bundle, and runs the application. You typically only call this function once, from your application's **main** function, which is usually generated automatically.

NSAvailableWindowDepths

SUMMARY This function returns the available NSWindowDepth values.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS const NSWindowDepth ***NSAvailableWindowDepths(**void**)**

DESCRIPTION **NSAvailableWindowDepths** returns a null-terminated array of NSWindowDepth values that specify which window depths are currently available. Window depth values are defined by the constants NSTwoBitGrayDepth, NSEightBitGrayDepth, and so on.

NSBeep

SUMMARY This function plays the system beep.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSBeep(**void**)**

DESCRIPTION This function plays the system beep. Users can select a sound to be played as the system beep. On a Macintosh, for example, you can change sounds with the Sound control panel.

NSBestDepth

SUMMARY This function attempts to return a window depth adequate for the specified parameters.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS NSWindowDepth **NSBestDepth(**NSString **colorSpace*, int *bps*, int *bpp*, BOOL *planar*, BOOL **exactMatch***)**

DESCRIPTION **NSBestDepth** returns a window depth deep enough for the given number of colors in *colorSpace*, bits per sample specified by *bps*, bits per pixel specified by *bpp*, and whether planar as specified by *planar*. Upon return, the variable pointed to by *exactMatch* is YES if the window depth can accommodate all of the values specified by the parameters, NO if it can't.

NSBitsPerPixelFromDepth

SUMMARY This function returns the bits per pixel for the specified window depth.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS int **NSBitsPerPixelFromDepth(**NSWindowDepth *depth***)**

DESCRIPTION **NSBitsPerPixelFromDepth** returns the number of bits per pixel for the window depth specified by *depth*.

SEE ALSO **NSBitsPerSampleFromDepth**

NSBitsPerSampleFromDepth

SUMMARY This function returns the bits per sample for the specified window depth.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS int **NSBitsPerSampleFromDepth(**NSWindowDepth *depth***)**

DESCRIPTION **NSBitsPerSampleFromDepth** returns the number of bits per sample (bits per pixel in each color component) for the window depth specified by *depth*.

SEE ALSO **NSBitsPerPixelFromDepth**

NSColorSpaceFromDepth

SUMMARY This function returns the name of the color space corresponding to the passed window depth.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS NSString ***NSColorSpaceFromDepth(**NSWindowDepth *depth***)**

DESCRIPTION Returns the color space name for the specified *depth*. For example, the returned color space name can be NSCalibratedRGBColorSpace, NSDeviceCMYKColorSpace, or so on.

NSConvertGlobalToWindowNumber

SUMMARY This function converts a global window number to a local window number.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSConvertGlobalToWindowNumber(**int *globalNum***,** unsigned int ***winNum***)**

DESCRIPTION In rare cases, two or more applications may need to refer to the same window. To pass a window number to another application, an application uses the global window number, which has been automatically assigned by the Window Server, rather than the local window number, which is assigned by the application.

An application uses the **NSConvertGlobalToWindowNumber** function to convert a window number from global to local. Given a global window number in *globalNum*, it returns the corresponding local window number in the location specified by *winNum*.

SEE ALSO **NSConvertWindowNumberToGlobal**

NSConvertGlyphsToPackedGlyphs

SUMMARY This function prepares a set of glyphs for processing by character-based routines.

DECLARED IN AppKit/NSFont.h

SYNOPSIS int **NSConvertGlyphsToPackedGlyphs**(NSGlyph **glBuf*, int *count*, NSMultibyteGlyphPacking *packing*, char **packedGlyphs*)

DESCRIPTION This function takes a buffer of glyphs, specified in the *glBuf* parameter, and packs them into a condensed character array. The character array is returned in the *packedGlyphs* parameter, which should have enough space for at least ((4 * *count*) + 1) bytes to guarantee that the packed glyphs fit. The *count* parameter specifies the number of glyphs in *glBuf*. The *packing* parameter specifies how the glyphs are currently packed.

NSConvertWindowNumberToGlobal

SUMMARY This function converts a local window number to a global window number.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSConvertWindowNumberToGlobal(**int *winNum*, unsigned int **globalNum***)*

DESCRIPTION In rare cases, two or more applications may need to refer to the same window. To pass a window number to another application, an application uses the global window number, which has been automatically assigned by the Window Server, rather than the local window number, which is assigned by the application.

NSConvertWindowNumberToGlobal takes the local window number and places the corresponding global window number in the variable specified by *globalNum*. This global number can then be passed to other applications that need access to the window.

SEE ALSO **NSConvertGlobalToWindowNumber**

NSCopyBitmapFromGState

SUMMARY This function copies a bitmap image to a destination rectangle.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSCopyBitmapFromGState(**int *srcGState***,** NSRect *srcRect***,** NSRect *destRect***)**

DESCRIPTION This function copies the pixels in the rectangle *srcRect* to the rectangle *destRect*. The source rectangle is defined in the graphics state designated by *srcGState*. The destination is defined in the current graphics state.

SEE ALSO **NSCopyBits**

NSCopyBits

SUMMARY This function copies a bitmap image to the location specified by a destination point.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSCopyBits(**int *srcGState***,** NSRect *srcRect***,** NSPoint *destPoint***)**

DESCRIPTION This function copies the pixels in the rectangle specified by *srcRect* to the location specified by *destPoint*. The source rectangle is defined in the graphics state designated by *srcGState*. If *srcGState* is NSNullObject, the current graphics state is assumed. The *destPoint* destination is defined in the current graphics state.

SEE ALSO **NSCopyBitmapFromGState**

NSCountWindows

SUMMARY This function counts the number of on-screen windows belonging to an application.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSCountWindows(**int **count***)**

DESCRIPTION **NSCountWindows** counts the number of on-screen windows belonging to the application; it returns the number by reference in the *count* parameter.

SEE ALSO **NSWindowList**

NSCreateFileContentsPboardType

SUMMARY This function returns a pasteboard type based on the passed file type.

DECLARED IN AppKit/NSPasteboard.h

SYNOPSIS NSString ***NSCreateFileContentsPboardType(**NSString ****fileType*)**

DESCRIPTION **NSCreateFileContentsPboardType** returns an NSString to a pasteboard type representing a file's contents based on the supplied string *fileType*. *fileType* should generally be the extension part of a file name. The conversion from a named file type to a pasteboard type is simple; no mapping to standard pasteboard types is attempted.

SEE ALSO **NSCreateFilenamePboardType, NSGetFileType, NSGetFileTypes**

NSCreateFilenamePboardType

SUMMARY This function returns a pasteboard type based on the passed file type.

DECLARED IN AppKit/NSPasteboard.h

SYNOPSIS NSString ***NSCreateFilenamePboardType(**NSString ****fileType*)**

DESCRIPTION **NSCreateFilenamePboardType** returns an NSString to a pasteboard type representing a file name based on the supplied string *fileType*.

SEE ALSO **NSCreateFileContentsPboardType, NSGetFileType, NSGetFileTypes**

NSDottedFrameRect

SUMMARY << Description forthcoming >>

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSDottedFrameRect**(NSRect *aRect*)

DESCRIPTION << Description forthcoming >>

NSDrawBitmap

SUMMARY This function draws a bitmap image.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSDrawBitmap(**const NSRect *rect*, int *pixelsWide*, int *pixelsHigh*, int *bitsPerSample*, int *samplesPerPixel*, int *bitsPerPixel*, int *bytesPerRow*, BOOL *isPlanar*, BOOL *hasAlpha*, NSColorSpace *colorSpace*, const unsigned char *const *data*[5]**)**

Warning: This function is marginally obsolete. Most applications are better served using the NSBitmapImageRep class to read and display bitmap images.

DESCRIPTION The **NSDrawBitmap** function renders an image from a bitmap, binary data that describes the pixel values for the image (this function replaces **NSImageBitmap**).

NSDrawBitmap renders a bitmap image using an appropriate PostScript operator—**image**, **colorimage**, or **alphaimage**. It puts the image in the rectangular area specified by its first argument, *rect*; the rectangle is specified in the current coordinate system and is located in the current window. The next two arguments, *pixelsWide* and *pixelsHigh*, give the width and height of the image in pixels. If either of these dimensions is larger or smaller than the corresponding dimension of the destination rectangle, the image will be scaled to fit.

The remaining arguments to **NSDrawBitmap** describe the bitmap data, as explained in the following paragraphs.

bitsPerSample is the number of bits per sample for each pixel and *samplesPerPixel* is the number of samples per pixel. *bitsPerPixel* is based on *samplesPerPixel* and the configuration of the

bitmap: if the configuration is planar, then the value of *bitsPerPixel* should equal the value of *bitsPerSample*; if the configuration isn't planar (is meshed instead), *bitsPerPixel* should equal *bitsPerSample * samplesPerPixel*.

bytesPerRow is calculated in one of two ways, depending on the configuration of the image data (data configuration is described below). If the data is planar, *bytesPerRow* is (7 + (*pixelsWide * bitsPerSample*)) / 8. If the data is meshed, *bytesPerRow* is (7 + (*pixelsWide * bitsPerSample * samplesPerPixel*)) / 8.

A sample is data that describes one component of a pixel. In an RGB color system, the red, green, and blue components of a color are specified as separate samples, as are the cyan, magenta, yellow, and black components in a CMYK system. Color values in a gray scale are a single sample. Alpha values that determine transparency and opaqueness are specified as a coverage sample separate from color. In bitmap images with alpha, the color (or gray) components have to be premultiplied with the alpha. This is the way images with alpha are displayed, this is the way they are read back, and this is the way they are stored in TIFFs.

isPlanar refers to the way data is configured in the bitmap. This flag should be set YES if a separate data channel is used for each sample. The function provides for up to five channels, *data1*, *data2*, *data3*, *data4*, and *data5*. It should be set NO if sample values are interwoven in a single channel (meshed); all values for one pixel are specified before values for the next pixel.

Figure 1 illustrates these two ways of configuring data.

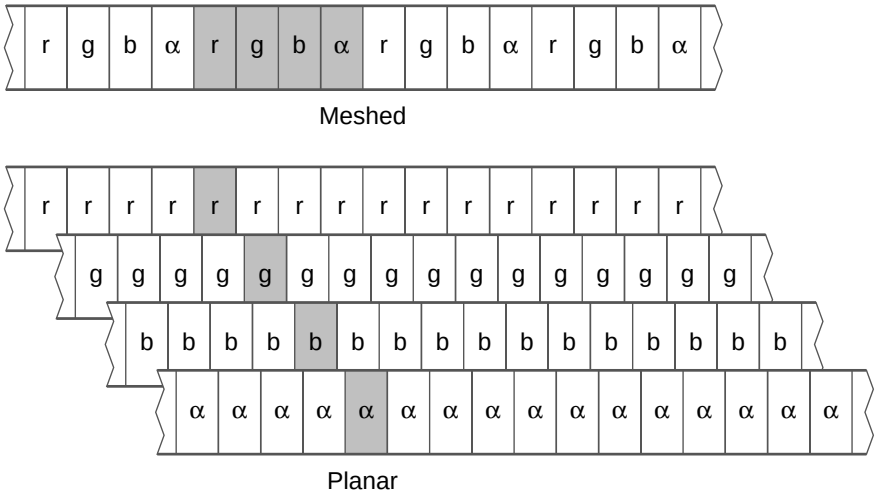


Figure 1. Planar and Meshed Configurations

As shown in the illustration, color samples (rgb) precede the coverage sample (α) in both configurations.

Gray-scale windows store pixel data in planar configuration; color windows store it in meshed configuration. **NSDrawBitmap** can render meshed data in a planar window, or planar data in a meshed window. However, it's more efficient if the image has a depth (*bitsPerSample*) and configuration (*isPlanar*) that matches the window.

hasAlpha indicates whether the image contains alpha. If it does, the number of samples should be 1 greater than the number of color components in the model (e.g., 4 for RGB).

colorSpace can be NS_CustomColorSpace, indicating that the image data is to be interpreted according to the current color space in the PostScript graphics state. This allows for imaging using custom color spaces. The image parameters supplied as the other arguments should match what the color space is expecting.

If the image data is planar, *data*[0] through *data*[*samplesPerPixel*-1] point to the planes; if the data is meshed, only *data*[0] needs to be set.

NSDrawButton, NSDrawGrayBezel, NSDrawGroove, NSDrawTiledRects, NSDrawWhiteBezel, NSFrameRect, NSFrameRectWithWidth

SUMMARY Draw a bordered rectangle

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS

void **NSDrawButton(**const NSRect *aRect*, const NSRect *clipRect***)**
void **NSDrawGrayBezel(**const NSRect *aRect*, const NSRect *clipRect***)**
void **NSDrawGroove(**const NSRect *aRect*, const NSRect *clipRect***)**
void **NSDrawWhiteBezel(**const NSRect *aRect*, const NSRect *clipRect***)**
NSRect **NSDrawTiledRects(**NSRect *aRect*, const NSRect *clipRect*, const NSRectEdge **sides*, const float **grays*, int *count***)**
void **NSFrameRect(**const NSRect *aRect***)**
void **NSFrameRectWithWidth(**const NSRect *aRect*, NSCoord *frameWidth***)**

DESCRIPTION These functions draw rectangles with borders. **NSDrawButton** draws the rectangle used to signify a user-interface button, **NSDrawTiledRects** is a generic function that can be used to draw different types of borders, and the other functions provide ready-made bezeled, grooved, or line

borders. These borders can be used to outline an area or to give rectangles the effect of being recessed from or elevated above the surface of the screen, as shown in Figure 2.

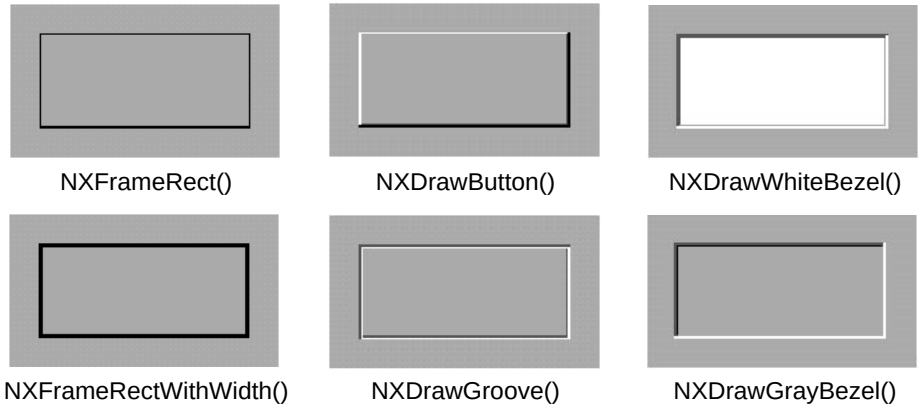


Figure 2. Rectangle Borders

Each function's first argument specifies the rectangle within which the border is to be drawn in the current coordinate system. Since these functions are often used to draw the border of a View, this rectangle will typically be that View's bounds rectangle. Some of the functions also take a clipping rectangle; only those parts of *aRect* that lie within the clipping rectangle will be drawn.

As its name suggests, **NSDrawWhiteBezel** fills in its rectangle with white; **NSDrawButton**, **NSDrawGrayBezel**, and **NSDrawGroove** use light gray. These functions are designed for rectangles that are defined in unscaled, unrotated coordinate systems (that is, where the y-axis is vertical, the x-axis is horizontal, and a unit along either axis is equal to one screen pixel). The coordinate system can be either flipped or unflipped. The sides of the rectangle should lie on pixel boundaries.

NSFrameRect and **NSFrameRectWithWidth** draw a frame around the inside of a rectangle in the current color. **NSFrameRect** draws a frame with a width equal to 1.0 in the current coordinate system; **NSFrameRectWithWidth** allows you to set the width of the frame. Since the frame is drawn inside the rectangle, it will be visible even if drawing is clipped to the rectangle (as it would be if the rectangle were a View object). These functions work best if the sides of the rectangle lie on pixel boundaries.

In addition to its *aRect* and *clipRect* arguments, **NSDrawTiledRects** takes three more arguments, which determine how thick the border is and what gray levels are used to form it. **NSDrawTiledRects** works through the Foundation framework's **NSDivideRect** function to take successive 1.0–unit-wide slices from the sides of the rectangle specified by the *sides* argument. Each slice is then drawn using the corresponding gray level from *grays*. **NSDrawTiledRects**

makes and draws these slices *count* number of times. **NSDivideRect** returns a pointer to the rectangle after the slice has been removed; therefore, if a side is used more than once, the second slice is made inside the first. This also makes it easy to fill in the rectangle inside of the border.

In the following example, **NSDrawTiledRects** draws a bezeled border consisting of a 1.0–unit-wide white line at the top and on the left side, and a 1.0–unit-wide dark-gray line inside a 1.0–unit-wide black line on the other two sides. The rectangle inside this border is filled in using light gray.

```
int    mySides[] = {NX_YMIN, NX_XMAX, NX_YMAX, NX_XMIN,
                    _YMIN, NX_XMAX};
float  myGrays[] = {NS_BLACK, NS_BLACK, NS_WHITE, NS_WHITE,
                    NS_DARKGRAY, NS_DARKGRAY};
NSRect *aRect;

NSDrawTiledRects(aRect, (NSRect *)0, mySides, myGrays, 6);
PSsetgray(NS_LIGHTGRAY);
PSrectfill(aRect->origin.x, aRect->origin.y,
          aRect->size.width, aRect->size.height);
```

As shown, **mySides** is an array that specifies sides of a rectangle; for example, NX_YMIN selects the side parallel to the x-axis with the smallest y-coordinate value. **myGrays** is an array that specifies the successive gray levels to be used in drawing parts of the border.

NSDrawTiledRects returns a pointer to the rectangle that lies within the border.

NSDrawColorTiledRects

SUMMARY << Description forthcoming >>

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS NSRect **NSDrawColorTiledRects**(NSRect *boundsRect*, NSRect *clipRect*, const NSRectEdge **sides*, NSColor ***colors*, int *count*)

DESCRIPTION << Description forthcoming >>

NSDrawDarkBezel

SUMMARY << Description forthcoming >>

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSDrawDarkBezel**(NSRect *aRect*, NSRect *clipRect*)

DESCRIPTION << Description forthcoming >>

NSDrawLightBezel

SUMMARY << Description forthcoming >>

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSDrawLightBezel**(NSRect *aRect*, NSRect *clipRect*)

DESCRIPTION << Description forthcoming >>

NSEraseRect

SUMMARY This function erases the passed rect by filling it with white.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSEraseRect(**const NSRect *aRect***)**

DESCRIPTION As its name suggests, **NSEraseRect** erases the rectangle referred to by its argument, filling it with white. It does not alter the current color.

SEE ALSO **NSHighlightRect, NSRectClip, NSRectClipList, NSRectFill, NSRectFillList, NSRectFillListWithColors, NSRectFillListWithGrays, NSUnionRect (Foundation Kit)**

NSEventMaskFromType

SUMMARY This function returns the event mask for the specified type.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS unsigned int **NSEventMaskFromType(**NSEventType *type***)**

DESCRIPTION **NSEventMaskFromType** returns the event mask corresponding to the specified *type* (an enumerated constant). The returned mask is equal to 1 left-shifted by *type* bits.

NSFrameLinkRect

SUMMARY This function draws a distinctive outline around linked data.

DECLARED IN AppKit/NSDataLinkManager.h

SYNOPSIS void **NSFrameLinkRect(**NSRect *aRect*, BOOL *isDestination***)**

DESCRIPTION **NSFrameLinkRect** draws a distinctive link outline just outside the rectangle specified by *aRect*. To draw an outline around a destination link, *isDestination* should be YES, otherwise it should be NO.

SEE ALSO **NSLinkFrameThickness**

NSGetAlertPanel

SUMMARY This function returns an alert panel.

DECLARED IN AppKit/NSPanel.h

SYNOPSIS id **NSGetAlertPanel(**NSString **title*, NSString **msg*, NSString **defaultButton*, NSString **alternateButton*, NSString **otherButton*, ...**)**

DESCRIPTION **NSGetAlertPanel** returns an NSAlert panel that can be used to set up a modal session. A modal session is useful for allowing the user to interrupt the program. During a modal session, you can perform activities while the panel is displayed and check at various points in your program whether the user has clicked one of the panel's buttons. The arguments for this function are the same as those for the **NSRunAlertPanel** function, but unlike that function, no button is displayed if *defaultButton* is **nil**.

To set up a modal session, send the Application object a **beginModalSession:for:** message with the panel returned by **NSGetAlertPanel** as its second argument. When you want to check if the user has clicked one of the panel's buttons, use **runModalSession:**. To end the modal session, use **endModalSession:**. When you're finished with the panel created by **NSGetAlertPanel**, you must free it by passing it to **NSReleaseAlertPanel**.

SEE ALSO **NSGetCriticalAlertPanel, NSGetInformationalAlertPanel, NSReleaseAlertPanel, NSRunAlertPanel, NSRunCriticalAlertPanel, NSRunInformationalAlertPanel**

NSGetCriticalAlertPanel

SUMMARY This function returns an alert panel to display a critical message.

DECLARED IN AppKit/NSPanel.h

SYNOPSIS id **NSGetCriticalAlertPanel(**NSString **title*, NSString **msg*, NSString **defaultButton*, NSString **alternateButton*, NSString **otherButton*, ...**)**

DESCRIPTION **NSGetCriticalAlertPanel** returns an NSAlert panel that can be used to set up a modal session. Unlike the **NSRunCriticalAlertPanel** function, no button is displayed if *defaultButton* is **nil**. When you're finished with the panel created by **NSGetCriticalAlertPanel**, you must free it by passing it to **NSReleaseAlertPanel**.

The arguments for this function are the same as those for the **NSRunAlertPanel** function. For more information on using a panel in a modal session, see **NSGetAlertPanel**.

SEE ALSO **NSGetAlertPanel, NSGetInformationalAlertPanel, NSReleaseAlertPanel, NSRunAlertPanel, NSRunCriticalAlertPanel, NSRunInformationalAlertPanel**

NSGetFileType

SUMMARY This function returns a file type based on the passed pasteboard type.

DECLARED IN AppKit/NSPasteboard.h

SYNOPSIS NSString ***NSGetFileType(**NSArray ***pboardType*)

DESCRIPTION **NSGetFileType** is the inverse of both **NSCreateFileContentsPboardType** and **NSCreateFilenamePboardType**. When passed a pasteboard type as returned by those functions, it returns the extension or file name from which the type was derived. It returns **nil** if *pboardType* isn't a pasteboard type created by those functions.

SEE ALSO **NSCreateFileContentsPboardType, NSCreateFilenamePboardType, NSGetFileTypes**

NSGetFileTypes

SUMMARY This function returns an array of file type based on the passed pasteboard types.

DECLARED IN AppKit/NSPasteboard.h

SYNOPSIS NSArray ***NSGetFileTypes(**NSArray ***pboardType*)

DESCRIPTION **NSGetFileTypes** accepts a null-terminated array of pointers to pasteboard types and returns a null-terminated array of the unique extensions and file names from the file-content and file-name types found in the input array. It returns **nil** if the input array contains no file-content or file-name types. The returned array is allocated and must be freed by the caller. The pointers in the return array point into strings passed in the input array.

SEE ALSO **NSCreateFileContentsPboardType, NSCreateFilenamePboardType, NSGetFileType**

NSGetInformationalAlertPanel

SUMMARY This function returns an alert panel to display an informational message.

DECLARED IN AppKit/NSPanel.h

SYNOPSIS id **NSGetInformationalAlertPanel(**NSString **title*, NSString **msg*, NSString **defaultButton*, NSString **alternateButton*, NSString **otherButton*, ...**)**

DESCRIPTION **NSGetInformationalAlertPanel** returns an NSAlert panel that can be used to set up a modal session. Unlike the **NSRunInformationalAlertPanel** function, no button is displayed if *defaultButton* is **nil**. When you're finished with the panel created by **NSGetInformationalAlertPanel**, you must free it by passing it to **NSReleaseAlertPanel**.

The arguments for this function are the same as those for the **NSRunAlertPanel** function. For more information on using a panel in a modal session, see **NSGetAlertPanel**.

SEE ALSO **NSGetAlertPanel, NSGetCriticalAlertPanel, NSReleaseAlertPanel, NSRunAlertPanel, NSRunCriticalAlertPanel, NSRunInformationalAlertPanel**

NSGetWindowServerMemory

SUMMARY This function returns the amount of memory being used by a context.

DECLARED IN AppKit/Application.h

SYNOPSIS int **NSGetWindowServerMemory(**DPSContext *context*, int **virtualMemory*, int **windowBackingMemory*, NSString ***windowDumpStream***)**

DESCRIPTION **NSGetWindowServerMemory** calculates the amount of Window Server memory being used at the moment by the given Window Server context. If **nil** is passed for the context, the current context is used. The amount of PostScript virtual memory used by the current context is returned in the **int** pointed to by *virtualMemory*; the amount of window backing store used by windows owned by the current context is returned in the **int** pointed to by *windowBackingMemory*. The sum of these two numbers is the amount of the Window Server's memory that this context is responsible for.

To calculate these numbers, **NSGetWindowServerMemory** uses the PostScript language operators **dumpwindows** and **vmstatus**. It takes some time to execute; thus, calling this function in normal operation is not recommended.

If a non–nil value is passed in for *windowDumpStream*, the information returned from the **dumpwindows** operator is echoed to the specified stream. This can be useful for finding out more about which windows are using up your storage.

Normally, **NSGetWindowServerMemory** returns 0. If **nil** is passed for context and there's no current DPS context, this function returns –1.

NSHighlightRect

SUMMARY This function highlights the passed rect by filling it with white.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSHighlightRect(**const NSRect *aRect***)**

DESCRIPTION **NSHighlightRect** uses the **compositerect** operator to highlight the rectangle referred to by its argument. Light gray becomes white, and white becomes light gray. This function must be called twice, once to highlight the rectangle and once to unhighlight it; the rectangle should not be left in its highlighted state. When not drawing on the screen, the compositing operation is replaced by one that fills the rectangle with light gray.

SEE ALSO **NSHighlightRect, NSRectClip, NSRectClipList, NSRectFill, NSRectFillList, NSRectFillListWithColors, NSRectFillListWithGrays, NSUnionRect (Foundation Kit)**

NSInterfaceStyleForKey

SUMMARY This function returns an interface style value for the specified key and responder.

DECLARED IN AppKit/NSInterfaceStyle.h

SYNOPSIS NSInterfaceStyle **NSInterfaceStyleForKey(**NSString **key*, NSResponder **responder***)**

DESCRIPTION You call the **NSInterfaceStyleForKey** function to determine an interface style based on a key and a responder, either of which may be **nil**. An NSInterfaceStyle value specifies the style in which an interface item, such as a button or a scrollbar, should be drawn. For example, a value of NSMacintoshInterfaceStyle indicates an item should be drawn in the Macintosh style. The enum values defined for NSInterfaceStyle are NSNoInterfaceStyle, NSNextStepInterfaceStyle, NSWindows95InterfaceStyle, and NSMacintoshInterfaceStyle. Note that **NSInterfaceStyleForKey** never returns NSNoInterfaceStyle.

The interface style value returned by **NSInterfaceStyleForKey** depends on several factors. If *responder* is not **nil** and if *responder* specifies an interface style other than NSNoInterfaceStyle, **NSInterfaceStyleForKey** returns the responder's style, and *key* is ignored.

Otherwise, if *key* is not **nil** and there is an interface style for *key* specified by the defaults system, **NSInterfaceStyleForKey** returns the interface style for *key* from the defaults system.

Finally, if *key* is **nil**, or if there is no interface style for *key* specified by the defaults system, **NSInterfaceStyleForKey** returns the global interface style specified by the defaults system.

The defaults system allows an application to customize its behavior to match a user's preferences.You can read about the defaults system in the documentation for NSUserDefaults.

NSLinkFrameThickness

SUMMARY This function returns the thickness of the outline around linked data.

DECLARED IN AppKit/NSDataLinkManager.h

SYNOPSIS float **NSLinkFrameThickness(**void**)**

DESCRIPTION **NSLinkFrameThickness** returns the thickness of the link outline around linked data so that the outline can be properly erased by the application, or for other purposes.

SEE ALSO **NSFrameLinkRect**

NSNumberOfColorComponents

SUMMARY This function returns the number of color components in the specified color space.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS int **NSNumberOfColorComponents(**NSString **colorSpaceName***)*

DESCRIPTION **NSNumberOfColorComponents** returns the number of color components in the color space whose name is provided by *colorSpaceName*.

NSPerformService

SUMMARY This function programmatically invokes a Services menu service.

DECLARED IN AppKit/Listener.h

SYNOPSIS BOOL **NSPerformService(**NSString **itemName*, NSPasteboard **pboard*)

DESCRIPTION **NSPerformService** allows an application to programmatically invoke a service found in its services menu. *itemName* is a Services menu item, in any language. If the requested service is from a submenu of the Services menu, *itemName* must contain a slash (for example, “Mail/Selection”). The Pasteboard *pboard* must contain the data required by the service, and when the function returns, *pboard* will contain the data supplied by the service provider.

NSPerformService returns YES if the service is successfully performed, NO otherwise.

NSPlanarFromDepth

SUMMARY This function returns whether the specified window depth is planar.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS BOOL **NSPlanarFromDepth(**NSWindowDepth *depth***)**

DESCRIPTION **NSPlanarFromDepth** returns YES if the specified window depth is planar and NO if it is not.

NSReadPixel

SUMMARY This function reads a pixel value at the specified location.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS **NSColor *NSReadPixel(NSPoint** *passedPoint***)**

DESCRIPTION **NSReadPixel** returns the color of the pixel at the given location. The *location* argument is taken in the current coordinate system—in other words, you must lock focus on the View that contains the pixel that you wish to query, and then pass the coordinate for the pixel in the View's coordinate system.

NSRectClip

SUMMARY This function modifies the current clipping path by intersecting it with the passed rect.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSRectClip(**NSRect *aRect***)**

DESCRIPTION **NSRectClip** intersects the current clipping path with the rectangle referred to by its argument, *aRect*, to determine a new clipping path. This function works through the **rectclip** operator. After computing the new clipping path, the current path is reset to empty.

SEE ALSO **NSEraseRect, NSHighlightRect, NSRectClipList, NSRectFill, NSRectFillList, NSRectFillListWithColors, NSRectFillListWithGrays, NSUnionRect (Foundation Kit)**

NSRectClipList

SUMMARY This function modifies the current clipping path by intersecting it with the passed rect.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSRectClipList(**const NSRect **rects*, int *count***)**

DESCRIPTION **NSRectClipList** takes an array of *count* number of rectangles, constructs a path that's the graphic union of those rectangles, and intersects that path with the current clipping path. This function works through the **rectclip** operator. After computing the new clipping path, the current path is reset to empty.

SEE ALSO **NSEraseRect, NSHighlightRect, NSRectClip, NSRectFill, NSRectFillList, NSRectFillListWithColors, NSRectFillListWithGrays, NSUnionRect (Foundation Kit)**

NSRectFill

SUMMARY This function fills the passed rect with the current color.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSRectFillList(**const NSRect *aRect***)**

DESCRIPTION **NSRectFill** fills the rectangle referred to by its argument with the current color. It works through the **rectfill** operator.

SEE ALSO **NSEraseRect, NSHighlightRect, NSRectClip, NSRectClipList, NSRectFillList, NSRectFillListWithColors, NSRectFillListWithGrays, NSUnionRect (Foundation Kit)**

NSRectFillList

SUMMARY This function fills the rectangles in the passed list with the current color.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSRectFillList(**const NSRect **rects*, int *count***)**

DESCRIPTION **NSRectFillList** fills a list of *count* rectangles with the current color. It works through the **rectfill** operator.

SEE ALSO **NSEraseRect, NSHighlightRect, NSRectClip, NSRectClipList, NSRectFill, NSRectFillListWithColors, NSRectFillListWithGrays, NSUnionRect (Foundation Kit)**

NSRectFillListWithColors

SUMMARY This function fills the rectangles in the passed list with the current color.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSRectFillListWithColors(**const NSRect **rects*, NSColor ***colors*, int *count***)**

DESCRIPTION **NSRectFillListWithColors** takes a list of *count* rectangles and a matching list of *count* color values. The first rectangle is filled with the first color, the second rectangle with the second color, and so on. There must be an equal number of rectangles and color values. The rectangles should

not overlap; the order in which they'll be filled can't be guaranteed. This function alters the current color of the current graphics state, setting it unpredictably to one of the values passed in *colors*.

SEE ALSO **NSEraseRect, NSHighlightRect, NSRectClip, NSRectClipList, NSRectFill, NSRectFillList, NSRectFillListWithGrays, NSUnionRect (Foundation Kit)**

NSRectFillListWithGrays

SUMMARY This function fills the rectangles in the passed list with the current color.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSRectFillListWithGrays(**const NSRect **rects*, const float **grays*, int *count***)**

DESCRIPTION **NSRectFillListWithGrays** takes a list of *count* rectangles and a matching list of *count* gray values. The first rectangle is filled with the first gray, the second rectangle with the second gray, and so on. There must be an equal number of rectangles and gray values. The rectangles should not overlap; the order in which they'll be filled can't be guaranteed. This function alters the current color of the current graphics state, setting it unpredictably to one of the values passed in *grays*.

SEE ALSO **NSEraseRect, NSHighlightRect, NSRectClip, NSRectClipList, NSRectFill, NSRectFillList, NSRectFillListWithColors, NSUnionRect (Foundation Kit)**

NSRegisterServicesProvider

SUMMARY This function registers a service provider.

DECLARED IN AppKit/NSApplication.h

SYNOPSIS void **NSRegisterServicesProvider(**id *provider*, NSString **name***)**

DESCRIPTION **NSRegisterServicesProvider** registers *provider* as a service provider and associates it with the specified *name*. *name* should be unique; it is the name by which the service is advertised to service requestors.

NSApplications shouldn't use this function. Instead, they should use NSApplication's **setServicesProvider:** method, passing a non-nil argument.

SEE ALSO **NSSetShowsServicesMenuItem, NSShowsServicesMenuItem, NSUnRegisterServicesProvider,**

NSReleaseAlertPanel

SUMMARY This function releases an attention panel.

DECLARED IN AppKit/NSPanel.h

SYNOPSIS void **NSReleaseAlertPanel(**id *alertPanel***)**

DESCRIPTION When you're finished with a panel created by a function such as **NSGetAlertPanel**, **NSGetCriticalAlertPanel**, or **NSGetInformationalAlertPanel**, you must free it by passing it to **NSReleaseAlertPanel**.

SEE ALSO **NSGetAlertPanel, NSRunAlertPanel, NSRunCriticalAlertPanel**

NSRunAlertPanel

SUMMARY This function creates an attention panel.

DECLARED IN AppKit/NSPanel.h

SYNOPSIS int **NSRunAlertPanel(**NSString **title*, NSString **msg*, NSString **defaultButton*, NSString **alternateButton*, NSString **otherButton*, ...**)**

DESCRIPTION **NSRunAlertPanel** creates an attention panel that alerts the user to some consequence of a requested action; the panel may also let the user cancel or modify the action. **NSRunAlertPanel** runs the panel in a modal event loop.

The first argument is the title of the panel, which should be at most a few words long. The default title is "Alert". The next argument is the message that's displayed in the panel. It can use

printf-style formatting characters; any necessary arguments should be listed at the end of the function's argument list (after the *otherButton* argument). For more information on formatting characters, see the Rhapsody man page for **printf**.

There are arguments to supply titles for up to three buttons, which will be displayed in a row across the bottom of the panel. The panel created by **NSRunAlertPanel** must have at least one button, which will have the symbol for the Return key; if you pass a nil title to the other two buttons, they won't be created. If **nil** is passed as the *defaultButton*, "OK" will be used as its title.

NSRunAlertPanel not only creates the panel, it puts the panel on screen and runs it using the **runModalFor:** method defined in the Application class. This method sets up a modal event loop that causes the panel to remain on screen until the user clicks one of its buttons. **NSRunAlertPanel** then removes the panel from the screen list and returns a value that indicates which of the three buttons the user clicked: NS_ALERTDEFAULT, NS_ALERTALTERNATE, or NS_ALERTOTHER. (If an error occurred while creating the panel, NS_ALERTERROR is returned.) For efficiency, **NSRunAlertPanel** creates the panel the first time it's called and reuses it on subsequent calls, reconfiguring it if necessary.

SEE ALSO **NSGetAlertPanel, NSGetCriticalAlertPanel, NSGetInformationalAlertPanel, NSReleaseAlertPanel, NSRunCriticalAlertPanel, NSRunInformationalAlertPanel**

NSRunCriticalAlertPanel

SUMMARY This function creates and runs a critical attention panel.

DECLARED IN AppKit/NSPanel.h

SYNOPSIS int **NSRunCriticalAlertPanel(**NSString **title*, NSString **msg*, NSString **defaultButton*, NSString **alternateButton*, NSString **otherButton*, ...**)**

DESCRIPTION **NSRunCriticalAlertPanel** creates an attention panel that alerts the user to some critical consequence of a requested action; the panel lets the user cancel the action and may allow the user to modify the action. It then runs the panel in a modal event loop.

The arguments for this function are the same as those for the **NSRunAlertPanel** function.

SEE ALSO **NSGetAlertPanel, NSGetCriticalAlertPanel, NSGetInformationalAlertPanel, NSReleaseAlertPanel, NSRunAlertPanel, NSRunInformationalAlertPanel**

NSRunInformationalAlertPanel

SUMMARY This function creates and runs an informational attention panel.

DECLARED IN AppKit/NSPanel.h

SYNOPSIS int **NSRunInformationalAlertPanel(**NSString **title*, NSString **msg*, NSString **defaultButton*, NSString **alternateButton*, NSString **otherButton*, ...**)**

DESCRIPTION **NSRunInformationalAlertPanel** creates an informational attention panel that provides information related to a requested action. It then runs the panel in a modal event loop.

The arguments for this function are the same as those for the **NSRunAlertPanel** function.

SEE ALSO **NSGetAlertPanel, NSGetCriticalAlertPanel, NSGetInformationalAlertPanel, NSReleaseAlertPanel, NSRunAlertPanel, NSRunCriticalAlertPanel**

NSSetShowsServicesMenuItem

SUMMARY This function specifies whether an item should be included in Services menus.

DECLARED IN AppKit/NSApplication.h

SYNOPSIS int **NSSetShowsServicesMenuItem(**NSString * *itemName*, BOOL *enabled***)**

DESCRIPTION **NSSetShowsServicesMenuItem** is used by a service-providing application to specify whether the Services menus of other applications will contain the *itemName* command; if so, users of those applications will be able to request services through that command. If *enabled* is YES, the Application Kit will build Services menus for other applications that include the *itemName* command. If *enabled* is NO, *item* won't appear in any application's Services menu. *itemName* should be the same, language-independent character string entered in the "Menu Item:" field of the services file.

Service-providing applications should let users decide whether the Services menus of other applications they use should include the *itemName* command.

NSSetShowsServicesMenuItem returns 0 if it's successful in enabling or disabling the *itemName* command, and a number other than 0 if not.

SEE ALSO **NSRegisterServicesProvider, NSShowsServicesMenuItem, NSUnRegisterServicesProvider,**

NSShowsServicesMenuItem

SUMMARY This function specifies whether a Services menu item is currently enabled.

DECLARED IN AppKit/NSApplication.h

SYNOPSIS BOOL **NSShowsServicesMenuItem(**NSString * *itemName***)**

DESCRIPTION **NSShowsServicesMenuItem** returns YES if *itemName* is currently enabled, and NO if it's not. *itemName* should be the same, language-independent character string entered in the "Menu Item: " field of the services file.

SEE ALSO **NSRegisterServicesProvider, NSSetShowsServicesMenuItem, NSUnRegisterServicesProvider,**

NSUnRegisterServicesProvider

SUMMARY This function unregisters a service provider.

DECLARED IN AppKit/NSApplication.h

SYNOPSIS void **NSUnRegisterServicesProvider(**NSString **name***name***)**

DESCRIPTION **NSUnRegisterServicesProvider** unregisters the object named by *name* as a service provider.

NSApplications shouldn't use this function. Instead, they should use NSApplication's **setServicesProvider:** method, passing a **nil** argument.

SEE ALSO **NSRegisterServicesProvider, NSSetShowsServicesMenuItem, NSShowsServicesMenuItem**

NSUpdateDynamicServices

SUMMARY This function causes the services information for the system to be updated.

DECLARED IN AppKit/NSApplication.h

SYNOPSIS void **NSUpdateDynamicServices(**void**)**

DESCRIPTION **NSUpdateDynamicServices** is used by a service-providing application to re-register the services it's willing to provide. To do this, you create a file with the extension ".service" and place it in the application's path or **~/Library/Services**. The content of the file is identical to a normal service file (see the "Other Features" section for a description of service file format). You then call this function.

It is only necessary to call **NSUpdateDynamicServices** if your program adds dynamic services to the system.

NSWindowList

SUMMARY Get information about an application's windows

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSWindowList(**int *size*, int *list*[]**)**

DESCRIPTION **NSWindowList** provides an ordered list of the application's on-screen windows. It fills the *list* array with up to *size* window numbers; the order of windows in the array is the same as their order in the Window Server's screen list (their front-to-back order on the screen). Use the count obtained by **NSCountWindows** to specify the size of the array for **NSWindowList**.

SEE ALSO **NSCountWindows**

Client Library Functions

Note: This section has not been updated and has not received recent technical review. It is included in this release to test the linkage between application development tools and on-line documentation. The information in this section should be considered at best preliminary and subject to change. An updated version of this file will be included in the next release.

DPSAddFD

SUMMARY Monitor a file descriptor

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSAddFD(**int *fd*, DPSFDProc *handler*, void **userData*, int *priority***)**

DESCRIPTION **DPSAddFD** registers the function *handler* to be called each time there is activity with the file specified by file descriptor *fd*. The function is called provided the following are true:

- The file descriptor *fd* must be valid and open; typically *fd* is generated through a call to **open**. There needn't be any data waiting to be read on *fd*.
- *priority*, an integer from 0 to 30, must be equal to or greater than the application's current priority threshold. See **DPSAddTimedEntry** for a further explanation.

DPSFDProc, *handler*'s defined type, takes the form

```
void *handler(int fd, void *userData)
```

where *fd* is the file descriptor that prompted the function call and *userData* is the same pointer that was passed as the third argument to **DPSAddFD**. The *userData* pointer is provided as a convenience, allowing you to pass arbitrary data to *handler*.

Typically, **DPSAddFD** is used to listen to a socket or pipe; it's rarely used to monitor a common file.

SEE ALSO **DPSAddPort, DPSAddTimedEntry, DPSRemoveFD**

DPSAddNotifyPortProc

SUMMARY Set the handler function for the notify port

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSAddNotifyPortProc(**DPSPortProc *handler*, void **userData***)*

DESCRIPTION **DPSAddNotifyPortProc** registers *handler* as the function that's called when a message arrives on the notify port, the unique port, created through the **task_notify** Mach function, on which notifications (such as port death) are sent. You don't need to create the notify port yourself; **DPSAddNotifyPortProc** creates it for you if it doesn't already exist.

DPSPortProc, *handler*'s defined type, takes the form

```
void *handler(msg_header_t *msg, void *userData)
```

where *msg* is a pointer to the message that was received at the port and *userData* is the same pointer that was passed as the second argument to **DPSAddNotifyPortProc**. The *userData* pointer is provided as a convenience, allowing you to pass arbitrary data to *handler*.

The notify port can have only one handler at a time; adding a handler removes the current one. You can remove the port's handler without specifying a new one with the **DPSRemoveNotifyPortProc** function. The function's argument must match the notify port's current handler.

SEE ALSO **DPSAddPort, DPSAddTimedEntry, DPSRemoveNotifyPortProc**

DPSAddPort

SUMMARY Monitor a Mach port

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSAddPort(**port_t *port*, DPSPortProc *handler*, int *maxMsgSize*, void **userData*, int *priority***)**

DESCRIPTION **DPSAddPort** registers the function *handler* to be called each time your application asks for an event or peeks at the event queue. The function is called provided the following are true:

- The Mach port *port* must be valid and it must hold a message waiting to be read.
- *priority*, an integer from 0 to 30, must be equal to or greater than the application's current priority threshold. See **DPSAddTimedEntry** for a further explanation.

DPSPortProc, *handler*'s defined type, takes the form

```
void *handler(msg_header_t *msg, void *userData)
```

where *msg* is a pointer to the message that was received at the port and *userData* is the same pointer that was passed as the fourth argument to **DPSAddPort**. The *userData* pointer is provided as a convenience, allowing you to pass arbitrary data to *handler*.

If, within *handler*, you want to call **msg_receive** to receive further messages at the port, you must first call **DPSRemovePort** to remove the port from the system's port set. (This is because your application can't receive messages from a port that's in a port set.) After your application is finished receiving messages directly from the port, it can call **DPSAddPort** to have the system continue to monitor the port.

The contents of the message buffer *msg*, as received by *handler*, are invalid after the function returns. If you need to save any of the information that you find.

The *maxMsgSize* argument is an integer that gives the size, in bytes, of the largest message you expect to receive.

SEE ALSO **DPSAddFD, DPSAddTimedEntry, DPSRemovePort**

DPSAddTimedEntry

SUMMARY Create a timed entry

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS DPSTimedEntry **DPSAddTimedEntry(**double *period*, DPSTimedEntryProc *handler*, void **userData*, int *priority***)**

DESCRIPTION **DPSAddTimedEntry** registers *handler* as a "timed entry," a function that's called repeatedly at a given time interval. *period* determines the number of seconds between calls to the timed entry. Whenever an application based on the Application Kit attempts to retrieve events from the event

queue, it also checks (depending on *priority*) to determine whether any timed entries are due to be called. *userData* is a pointer that you can use to pass some data to the timed entry.

The function registered as *handler* has the form:

```
void *handler(DPSTimedEntry tag, double now, void *userData)
```

where *tag* is the timed entry identifier returned by **DPSAddTimedEntry**, *now* is the number of seconds since some arbitrary point in the past, and *userData* is the pointer **DPSAddTimedEntry** received when this timed entry was installed.

An application's priority threshold can be set explicitly as an integer from 0 to 31 through a call to **DPSGetEvent** or **DPSPeekEvent**. It's against this threshold that *priority* is measured (note that *priority* can be no greater than 30—the additional threshold level, 31, is provided to disallow all inter-event function calls). However, if you're using the Application Kit, you should access the event queue through Application class methods such as **getNextEvent:**. Although some of these methods let you set the priority threshold explicitly, you typically invoke the methods that set it automatically. Such methods use only three priority levels:

Constant	Meaning
NX_BASETHRESHOLD	Normal execution
NX_RUNMODALTHRESHOLD	An attention panel is being run
NX_MODALRESPTHRESHOLD	A modal event loop is being run

When applicable, you should use these constants as the value for *priority*. For example, if you want *handler* to be called during normal execution, but not if an attention panel or a modal loop is running, then you would set *priority* to NX_BASETHRESHOLD.

RETURN **DPSAddTimedEntry** returns a number identifying the timed entry or –1 to indicate an error.

SEE ALSO **DPSRemoveTimedEntry**

DPSAsynchronousWaitContext

SUMMARY Proceed asynchronously while PostScript code is executed

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSAsynchronousWaitContext(**DPSContext *context*, DPSPingProc *handler*, void **userData***userData***)**

DESCRIPTION This function is similar to the more familiar **DPSWaitContext** functions, except that rather than wait for all PostScript code to execute, it returns immediately, allowing your application to proceed while the PostScript code is executed in the background. The DPSPingProc function *handler* is called (with *context* and *userData* as its two arguments) when all the PostScript code has been executed. The DPSPingProc function takes the form

```
void *handler(DPSContext context, void *userData);
```

Warning: Be careful when you use this function; you must not send more PostScript code while waiting for the handler to be called. In general, it's best to not make any demands on the Application Kit or the Client Library if you're waiting for an asynchronous handler to return.

DPSCreateContext

SUMMARY Create a PostScript execution context

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS DPSContext **DPSCreateContext(**const char **hostName*, const char **serverName*, DPSTextProc *textProc*, DPSErrorProc *errorProc***)**

DESCRIPTION **DPSCreateContext** establishes a connection with the Window Server and creates a PostScript execution context in it. The new context becomes the current context. The first argument, *hostName*, identifies the machine that's running the Window Server; the second argument, *serverName*, identifies the Window Server that's running on that machine. With these two arguments and the help of the Mach network server **nmserver**, the Mach port for the Window Server can be identified. If *hostName* is NULL, the network server on the local machine is queried

for the Window Server's port. If *serverName* is NULL, a default name for the Window Server is used.

The last two arguments, *textProc* and *errorProc*, refer to call-back functions (defined in the Client Library specification) that handle text returned from the Window Server and errors generated on either side of the connection.

For an application that's based on the Application Kit, you could create an additional context by making this call:

```
DPSContext c;

c = DPSCreateContext(NXGetDefaultValue([NXApp appName], "NXHost"),
                        XGetDefaultValue([NXApp appName], "NXPSName"),
                        NULL,
                        NULL);
```

This example queries the application's default values for the identity of the host machine and the Window Server. By doing this, the new context is created in the correct Window Server even if that Server is not on the same machine as the application process.

The context that **DPSCreateContext** creates allocates memory from the default allocation zone. Also, when there's difficulty creating the context, **DPSCreateContext** waits up to 60 seconds before raising an exception. If you want to change either of these parameters, use **DPSCreateContextWithTimeoutFromZone**. Its two additional arguments let you specify the zone for the context to use when allocating context-specific data and a timeout value in milliseconds.

This function returns the newly created DPSContext structure.

EXCEPTIONS

DPSCreateContext raises a dps_err_outOfMemory exception if it encounters difficulty allocating ports or other resources for the new context. It raises a dps_err_cantConnect exception if it can't return a context within the timeout period.

SEE ALSO

DPSCreateContextWithTimeoutFromZone, DPSCreateNonsecureContext, DPSCreateStreamContext

DPSCreateContextWithTimeoutFromZone

SUMMARY Create a PostScript execution context

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS DPSContext **DPSCreateContextWithTimeoutFromZone(**const char **hostName*, const char **serverName*, DPSTextProc *textProc*, DPSErrorProc *errorProc*, int *timeout*, NSZone **zone***)*

DESCRIPTION **DPSCreateContextWithTimeoutFromZone** is identical to DPSCreateContext except that it accepts two additional arguments that let you specify the zone to use when allocating context-specific data and a timeout value other than the default value of 60 seconds. Specify the new *timeout* value in milliseconds,

This function returns the newly created DPSContext structure.

EXCEPTIONS **DPSCreateContextWithTimeoutFromZone** raises a dps_err_outOfMemory exception if it encounters difficulty allocating ports or other resources for the new context. It raises a dps_err_cantConnect exception if it can't return a context within the timeout period.

SEE ALSO **DPSCreateContext, DPSCreateNonsecureContext, DPSCreateStreamContext**

DPSCreateNonsecureContext

SUMMARY Create a PostScript execution context

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS DPSContext **DPSCreateNonsecureContext(**const char **hostName*, const char **serverName*, DPSTextProc *textProc*, DPSErrorProc *errorProc*, int *timeout*, NSZone **zone***)*

DESCRIPTION **DPSCreateNonsecureContext** creates a "nonsecure" context in which you can use PostScript operators that are normally disallowed. The most significant of these are operators that let you write files.

Few programmers will need to call this function directly: The Application Kit manages contexts for programs based on the Kit. For example, when an application is launched, its Application object calls **DPSCreateContext** to create a context in the Window Server. Similarly, to print a View the Kit calls **DPSCreateStreamContext** to temporarily redirect PostScript code from the View to a stream.

This function returns the newly created DPSContext structure.

SEE ALSO **DPSCreateContext, DPSCreateContextWithTimeoutFromZone, DPSCreateStreamContext**

DPSCreateStreamContext

SUMMARY Create a PostScript execution context

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS DPSContext **DPSCreateStreamContext(**NXStream **stream*, int *debugging*, DPSProgramEncoding *progEnc*, DPSNameEncoding *nameEnc*, DPSErrorProc *errorProc***)**

DESCRIPTION **DPSCreateStreamContext** is similar to **DPSCreateContext**, except that the new context is actually a connection from the client application to a stream. This connection becomes the current context. PostScript code that the application generates is sent to the stream rather than to the Window Server. The first argument, *stream*, is a pointer to an NXStream structure, as created by **NXOpenFile** or **NXMapFile**. The *debugging* argument is intended for debugging purposes but is not currently implemented. *progEnc* and *nameEnc* specify the type of program and user-name encodings to be used for output to the stream. The last argument, *errorProc*, identifies the procedure that's called when errors are generated.

Few programmers will need to call this function directly: The Application Kit manages contexts for programs based on the Kit. For example, when an application is launched, its Application object calls **DPSCreateContext** to create a context in the Window Server. Similarly, to print a View the Kit calls **DPSCreateStreamContext** to temporarily redirect PostScript code from the View to a stream.

This function returns the newly created DPSContext structure.

SEE ALSO **DPSCreateContext, DPSCreateContextWithTimeoutFromZone, DPSCreateNonsecureContext**

DPSDefineUserObject

SUMMARY Create a user object

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS int **DPSDefineUserObject(**int *index***)**

DESCRIPTION **DPSDefineUserObject** associates *index* with the PostScript object that's on the top of the operand stack, thereby creating a user object (as defined by the PostScript language). If *index* is 0, the object is assigned the next available index number. The function returns the new index, which can then be passed to a **pswrap**-generated function that takes a user object.

Warning: To avoid coming into conflict with user objects defined by the Client Library or Application Kit, use **DPSDefineUserObject** rather than the PostScript operator **defineuserobject** or the single-operator functions **DPSdefineuserobject** and **PSdefineuserobject**.

RETURN **DPSDefineUserObject**, if successful in assigning an index, returns the index that the object was assigned. If unsuccessful, it returns 0.

SEE ALSO **DPSUndefineUserObject**

DPSDiscardEvents

SUMMARY Discard events from the Window Server

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSDiscardEvents(**DPSContext *context*, int *mask***)**

DESCRIPTION **DPSDiscardEvents**'s two parameters, *context* and *mask*, are the same as those for **DPSGetEvent** and **DPSPeekEvent**. **DPSDiscardEvents** removes from the application's event queue those records whose event types match *mask* and whose context matches *context*.

RETURN **DPSGetEvent** and **DPSPeekEvent** return 1 if they are successful in accessing an event record and 0 if they aren't.

SEE ALSO **DPSGetEvent, DPSPeekEvent, DPSAddFD, DPSAddPort, DPSAddTimedEntry, DPSPostEvent**

DPSDoUserPath

SUMMARY Send an encoded PostScript path to the Window Server

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSDoUserPath(**void **coords*, int *numCoords*, DPSNumberFormat *numType*, unsigned char **ops*, int *numOps*, void **bbox*, int *action***)**

DESCRIPTION **DPSDoUserPath** sends an encoded user path to the Window Server and then executes, upon that path, the operator specified by *action*. The use of this function, rather than the analogous step-by-step path construction, is encouraged; rendering an encoded path is much more efficient than executing the individual PostScript operators that would otherwise be needed.

An encoded user path consists of an array of coordinate values, a sequence of PostScript operators, and a bounding box specification. The values in the coordinate array are used as operands to the operators; the operands are distributed to the operators in the order that they're given. The resulting path must fit within the bounding box.

The coordinates, operators, and bounding box are given by the function's first five arguments:

- The array of coordinate values is given by *coords*.
- *numCoords* is the number of elements in *coords*.
- *numType* specifies the data type of the coordinates, as described below. All the values in *coords* must be of the same type.
- *ops* is the sequence of PostScript operators, represented by constants as listed below.
- The bounding box is defined by the four coordinate values that you pass as an array in the *bbox* argument. These are passed as operands to the **setbbox** operator. (If you don't supply a **setbbox** as part of the *ops* sequence, one is inserted for you.)

The following integer constants represent the data types that you can pass as the *numType* argument:

Constant	Meaning
dps_float	single-precision floating-point number
dps_long	32-bit integer
dps_short	8-bit integer

You can also specify 16- and 32-bit fixed-point real numbers. For 16-bit fixed-point numbers, use **dps_short** plus the number of bits in the fractional portion. For 32-bit fixed-point numbers, use **dps_long** plus the number of bits in the fractional portion.

These constants are provided for *ops*:

dps_setbbox
dps_moveto
dps_rmoveto
dps_lineto
dps_rlineto
dps_curveto
dps_rcurveto
dps_arc
dps_arcn
dps_arct
dps_closepath
dps_ucache

Once the user path has been constructed, the operator specified by *action* is executed. The value of *action* is an index into Display PostScript's encoded system names; the following constants, provided as a convenience, represent the most commonly used actions:

dps_uappend
dps_ufill
dps_ueofill
dps_ustroke
dps_ustrokepath
dps_inufill
dps_inueofill
dps_inustroke
dps_def
dps_put

The following program fragment demonstrates the use of **DPSDoUserPath** as it creates and strokes a user path (an isosceles triangle) within a bounding rectangle whose lower left corner is located at (0, 0) and whose width and height are 200.

```
short  coords[6] = {0, 0, 200, 0, 100, 200};
char  ops[4] = {dps_moveto, dps_lineto,dps_lineto, dps_closepath};
short  bbox[4] = {0, 0, 200, 200};

DPSDoUserPath(coords, 6, dps_short, ops, 4, bbox, dps_ustroke);
```

Note: If an application calls **DPSDoUserPath** with large values (~10,000-20,000) of *numCoords* and/or *numOps*, it may generate a Display PostScript error.

SEE ALSO **DPSDoUserPathWithMatrix**

DPSDoUserPathWithMatrix

SUMMARY Send an encoded PostScript path to the Window Server

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSDoUserPathWithMatrix(**void **coords*, int *numCoords*, DPSNumberFormat *numType*, unsigned char **ops*, int *numOps*, void **bbox*, int *action*, float *matrix*[6]**)**

DESCRIPTION **DPSDoUserPathWithMatrix** sends an encoded user path to the Window Server and then executes, upon that path, the operator specified by *action*. The use of this function, rather than the analogous step-by-step path construction, is encouraged; rendering an encoded path is much more efficient than executing the individual PostScript operators that would otherwise be needed.

An encoded user path consists of an array of coordinate values, a sequence of PostScript operators, and a bounding box specification. The values in the coordinate array are used as operands to the operators; the operands are distributed to the operators in the order that they're given. The resulting path must fit within the bounding box.

The coordinates, operators, and bounding box are given by the function's first five arguments:

- The array of coordinate values is given by *coords*.
- *numCoords* is the number of elements in *coords*.

- *numType* specifies the data type of the coordinates, as described below. All the values in *coords* must be of the same type.
- *ops* is the sequence of PostScript operators, represented by constants as listed below.
- The bounding box is defined by the four coordinate values that you pass as an array in the *bbox* argument. These are passed as operands to the **setbbox** operator. (If you don't supply a **setbbox** as part of the *ops* sequence, one is inserted for you.)

The following integer constants represent the data types that you can pass as the *numType* argument:

Constant	Meaning
dps_float	single-precision floating-point number
dps_long	32-bit integer
dps_short	8-bit integer

You can also specify 16- and 32-bit fixed-point real numbers. For 16-bit fixed-point numbers, use **dps_short** plus the number of bits in the fractional portion. For 32-bit fixed-point numbers, use **dps_long** plus the number of bits in the fractional portion.

These constants are provided for *ops*:

dps_setbbox
dps_moveto
dps_rmoveto
dps_lineto
dps_rlineto
dps_curveto
dps_rcurveto
dps_arc
dps_arcn
dps_arct
dps_closepath
dps_ucache

Once the user path has been constructed, the operator specified by *action* is executed. The value of *action* is an index into Display PostScript's encoded system names; the following constants, provided as a convenience, represent the most commonly used actions:

dps_uappend
dps_ufill
dps_ueofill
dps_ustroke
dps_ustrokepath
dps_inufill
dps_inueofill
dps_inustroke
dps_def
dps_put

DPSDoUserPathWithMatrix's *matrix* argument represents the transformation matrix operand used by the **ustroke**, **inustroke**, and **ustrokepath** operators. If *matrix* is NULL, the argument is ignored.

DPSFlush

SUMMARY Send PostScript data to the Window Server

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSFlush**

DESCRIPTION **DPSFlush** flushes the application's output buffer, forcing any buffered PostScript code or data to the Window Server.

SEE ALSO **DPSSendEOF**

DPSGetEvent

SUMMARY Access events from the Window Server

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS int **DPSGetEvent(**DPSContext *context*, NXEvent **anEvent*, int *mask*, double *timeout*, int *threshold***)**

DESCRIPTION **DPSGetEvent** and **DPSPeekEvent** are macros that access event records in an application's event queue. These routines are provided primarily for programs that don't use the Application Kit. An application based on the Kit should use the corresponding Application class methods (such as **getNextEvent:** and **peekNextEvent:into:**) so that it can be journaled. **DPSDiscardEvents** removes all event records of a specified type from the queue.

DPSGetEvent and **DPSPeekEvent** differ only in how they treat the accessed event record. **DPSGetEvent** removes the record from the queue after making its data available to the application; **DPSPeekEvent** leaves the record in the queue.

DPSGetEvent and **DPSPeekEvent** take the same parameters. The first, *context*, represents a PostScript execution context within the Window Server. Virtually all applications have only one execution context, which can be returned using **DPSGetCurrentContext**. Applications having more than one execution context can use the constant DPS_ALLCONTEXTS to access events from all contexts belonging to them.

The second argument, *anEvent*, is a pointer to an event record. If **DPSGetEvent** or **DPSPeekEvent** is successful in accessing an event record, the record's data is copied into the storage referred to by *anEvent*.

mask determines the types of events sought. See the section "Types and Constants" for a list of the constants that represent the event type masks. To check for more than one type of event, you combine individual constants using the bitwise OR operator.

If an event matching the event mask isn't available in the queue, **DPSGetEvent** or **DPSPeekEvent** waits until one arrives or until *timeout* seconds have elapsed, whichever occurs first. The value of *timeout* can be in the range of 0.0 to NX_FOREVER. If *timeout* is 0.0, the routine returns an event only if one is waiting in the queue when the routine asks for it. If *timeout* is NX_ FOREVER, the routine waits until an appropriate event arrives before returning.

The last argument, *threshold*, is an integer in the range 0 through 31 that determines which other services may be provided during a call to **DPSGetEvent** or **DPSPeekEvent**.

Requests for services are registered by the functions **DPSAddTimedEntry**, **DPSAddPort**, and **DPSAddFD**. Each of these functions takes an argument specifying a priority level. If this level is equal to or greater than *threshold*, the service is provided before **DPSGetEvent** or **DPSPeekEvent** returns.

RETURN **DPSGetEvent** returns 1 if it is successful in accessing an event record and 0 if it isn't.

SEE ALSO **DPSPeekEvent, DPSDiscardEvents, DPSAddFD, DPSAddPort, DPSAddTimedEntry, DPSPostEvent**

DPSInterruptContext

Warning: This function is unimplemented in the NEXTSTEP version of the Client Library.

DPSNameFromTypeAndIndex

SUMMARY Access the system and user name tables

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS const char ***DPSNameFromTypeAndIndex(**short *type*, int *index***)**

DESCRIPTION **DPSNameFromTypeAndIndex** returns the text associated with *index* from the system or user name table. If *type* is –1, the text is returned from the system name table; if *type* is 0, it's returned from the user name table.

The name tables are used primarily by the Client Library and **pswrap**; few programmers will access them directly.

RETURN This function returns a read-only character string.

DPSPeekEvent

SUMMARY Access events from the Window Server

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS int **DPSPeekEvent(**DPSContext *context*, NXEvent **anEvent*, int *mask*, double *timeout*, int *threshold***)**

DESCRIPTION **DPSGetEvent** and **DPSPeekEvent** are macros that access event records in an application's event queue. These routines are provided primarily for programs that don't use the Application Kit. An application based on the Kit should use the corresponding Application class methods (such as **getNextEvent:** and **peekNextEvent:into:**) so that it can be journaled. **DPSDiscardEvents** removes all event records of a specified type from the queue.

DPSGetEvent and **DPSPeekEvent** differ only in how they treat the accessed event record. **DPSGetEvent** removes the record from the queue after making its data available to the application; **DPSPeekEvent** leaves the record in the queue.

DPSGetEvent and **DPSPeekEvent** take the same parameters. The first, *context*, represents a PostScript execution context within the Window Server. Virtually all applications have only one execution context, which can be returned using **DPSGetCurrentContext**. Applications having more than one execution context can use the constant DPS_ALLCONTEXTS to access events from all contexts belonging to them.

The second argument, *anEvent*, is a pointer to an event record. If **DPSGetEvent** or **DPSPeekEvent** is successful in accessing an event record, the record's data is copied into the storage referred to by *anEvent*.

mask determines the types of events sought. See the section "Types and Constants" for a list of the constants that represent the event type masks. To check for more than one type of event, you combine individual constants using the bitwise OR operator.

If an event matching the event mask isn't available in the queue, **DPSGetEvent** or **DPSPeekEvent** waits until one arrives or until *timeout* seconds have elapsed, whichever occurs first. The value of *timeout* can be in the range of 0.0 to NX_FOREVER. If *timeout* is 0.0, the routine returns an event only if one is waiting in the queue when the routine asks for it. If *timeout* is NX_ FOREVER, the routine waits until an appropriate event arrives before returning.

The last argument, *threshold*, is an integer in the range 0 through 31 that determines which other services may be provided during a call to **DPSGetEvent** or **DPSPeekEvent**.

Requests for services are registered by the functions **DPSAddTimedEntry**, **DPSAddPort**, and **DPSAddFD**. Each of these functions takes an argument specifying a priority level. If this level is equal to or greater than *threshold*, the service is provided before **DPSGetEvent** or **DPSPeekEvent** returns.

RETURN **DPSPeekEvent** returns 1 if it is successful in accessing an event record and 0 if it isn't.

SEE ALSO **DPSGetEvent, DPSDiscardEvents, DPSAddFD, DPSAddPort, DPSAddTimedEntry, DPSPostEvent**

DPSPostEvent

SUMMARY Create an event

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS int **DPSPostEvent(**NXEvent **anEvent*, int *atStart***)**

DESCRIPTION **DPSPostEvent** lets you add an event record to your application's event queue without involving the Window Server. *anEvent* is a pointer to the event record to be added. *atStart* specifies where the new record will be placed in relation to any other records in the queue. If *atStart* is TRUE, the event is posted in front of all others and so will be the next one your application receives. If *atStart* is FALSE, the event is posted behind all others and so won't be returned until events that precede it are processed.

You can free, reuse, or otherwise mangle *anEvent* after you've posted it without fear of corrupting the posted record. **DPSEvent** copies the record it receives and posts the copy.

Note that event records you post using **DPSPostEvent** aren't filtered by an event filter function set with **DPSSetEventFunc**.

RETURN **DPSPostEvent** returns 0 if successful in posting the event record; it returns –1 if unsuccessful in posting the record because the event queue is full.

SEE ALSO **DPSSetEventFunc**

DPSPrintError

SUMMARY Print error messages

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSPrintError(**FILE **fp*, const DPSBinObjSeq *error***)**

DESCRIPTION **DPSPrintError** and **DPSPrintErrorToStream** format and print error messages received from a PostScript execution context in the Window Server. The error message is extracted from the binary object sequence *error*. **DPSPrintError** prints the error message to the file identified by *fp*; **DPSPrintErrorToStream** prints the error message to *stream*.

You rarely need to call this function directly. However, if you reset the error handler for a PostScript execution context, the new handler you install could use this function to process errors that it receives.

SEE ALSO **DPSPrintErrorToStream**

DPSPrintErrorToStream

SUMMARY Print error messages

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSPrintErrorToStream(**NXStream **stream*, const DPSBinObjSeq *error***)**

DESCRIPTION **DPSPrintError** and **DPSPrintErrorToStream** format and print error messages received from a PostScript execution context in the Window Server. The error message is extracted from the binary object sequence *error*. **DPSPrintError** prints the error message to the file identified by *fp*; **DPSPrintErrorToStream** prints the error message to *stream*.

You rarely need to call this function directly. However, if you reset the error handler for a PostScript execution context, the new handler you install could use this function to process errors that it receives.

SEE ALSO **DPSPrintError**

DPSRemoveFD

SUMMARY Stop monitoring a file descriptor

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSRemoveFD(**int *fd***)**

DESCRIPTION **DPSRemoveFD** removes the specified file descriptor from the list of those that the application will check.

SEE ALSO **DPSAddFD**

DPSRemoveNotifyPortProc

SUMMARY Removes the handler function for the notify port

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSRemoveNotifyPortProc(**DPSPortProc *handler***)**

DESCRIPTION Removes the notify port's handler without specifying a new one. The *handler* argument must match the notify port's current handler.

SEE ALSO **DPSAddNotifyPortProc**

DPSRemovePort

SUMMARY Remove the Mach port being monitored.

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSRemovePort(**port_t *port***)**

DESCRIPTION **DPSRemovePort** removes the specified Mach port from the list of those that the application will check.

SEE ALSO **DPSAddPort**

DPSRemoveTimedEntry

SUMMARY Create a timed entry

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSRemoveTimedEntry(**DPSTimedEntry *tag***)**

DESCRIPTION **DPSRemoveTimedEntry** removes a previously registered timed entry.

SEE ALSO **DPSAddTimedEntry**

DPSResetContext

Warning: This function is unimplemented in the NEXTSTEP version of the Client Library.

DPSSendEOF

SUMMARY Send PostScript data to the Window Server

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSSendEOF(**DPSContext *context***)**

DESCRIPTION **DPSSendEOF** sends a PostScript end-of-file marker to the given context. The connection to the context isn't closed or disturbed in any way by this function.

SEE ALSO **DPSFlush**

DPSSetDeadKeysEnabled

SUMMARY Allow dead key processing for a context's events

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSSetDeadKeysEnabled(**DPSContext *context*, int *flag***)**

DESCRIPTION **DPSSetDeadKeysEnabled** turns dead key processing on or off for *context*. If flag is 0, dead key processing is turned off; otherwise, it's turned on (the default).

Dead key processing is a technique for extending the range of characters that can be entered from the keyboard. In NEXTSTEP, it provides one way for users to enter accented characters. For example, a user can type Alternate-e followed by the letter "e" to produce the letter "é". The first keyboard input, Alternate-e, seems to have no effect—it's the "dead key". However, it signals Client Library routines that it and the following character should be analyzed as a pair. If, within NEXTSTEP, the pair of characters has been associated with a third character, a keyboard event record representing the third character is placed in the application's event queue, and the first two event records are discarded. If there is no such association between the two characters, the two event records are added to the event queue.

DPSSetEventFunc

SUMMARY Set the function that filters events

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS DPSEventFilterFunc **DPSSetEventFunc(**DPSContext *context*, DPSEventFilterFunc *func***)**

DESCRIPTION **DPSSetEventFunc** establishes the function *func* as the function to be called when an event record is returned from the PostScript context *context* in the Window Server. The registered function is called before the event record is put in the event queue. If the registered function returns 0, the record is discarded. If the registered function returns 1, the record is passed on for further processing.

Only event records coming from the Window Server are filtered by the registered function. Records that you post to the event queue using **DPSPostEvent** aren't affected.

A DPSEventFilterFunc function takes the following form:

```
int *func(NXEvent *anEvent)
```

RETURN **DPSSetEventFunc** returns a pointer to the previously registered event function. This lets you chain together the current and previous event functions.

SEE ALSO **DPSPostEvent**

DPSSetTracking

SUMMARY Coalesce mouse events

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS int **DPSSetTracking(**int *flag***)**

DESCRIPTION **DPSSetTracking** turns mouse event-coalescing on or off for the current context. If *flag* is 0, coalescing is turned off; otherwise, it's turned on (the default).

Event coalescing is an optimization that's useful when tracking the mouse. When the mouse is moved, numerous events flow into the event queue. To reduce the number of events awaiting removal by the application, adjacent mouse-moved events are replaced by the most recent event of the contiguous group. The same is done for left and right mouse-dragged events, with the addition that a mouse-up event replaces mouse-dragged events that come before it in the queue.

RETURN **DPSSetTracking** returns the previous state of the event-coalescing switch.

DPSStartWaitCursorTimer

SUMMARY Initiate a count down for the wait cursor

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSStartWaitCursorTimer**

DESCRIPTION **DPSStartWaitCursorTimer** triggers the mechanism that displays a wait cursor when an application is busy and can't respond to user input. In most cases, wait cursor support is automatic: You'll only need to call this function if your application starts a time-consuming operation that's not initiated by a user-generated event.

Client Library routines and the Window Server cooperate to display the wait cursor whenever more than a preset amount of time elapses between the time an application takes an event record from the event queue and the time the application is again ready to consume events. However, when an application starts an operation that isn't initiated by an event—such as one caused by receiving a Mach message or by processing data from a file (see **DPSAddPort** and **DPSAddFD**)—the wait cursor mechanism is bypassed. To ensure proper wait cursor behavior in these cases, call **DPSStartWaitCursorTimer** before beginning the time-consuming operation.

SEE ALSO **DPSAddFD, DPSAddPort**

DPSSynchronizeContext

SUMMARY Synchronize a context with your application

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS int **DPSSynchronizeContext(**DPSContext *context*, int *flag***)**

DESCRIPTION **DPSSynchronizeContext** causes **DPSWaitContext** to be called after each **pswrap** function is called, thus synchronizing the PostScript context with your application.

DPSTraceContext

SUMMARY Trace data and events

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS int **DPSTraceContext(**DPSContext *context*, int *flag***)**

DESCRIPTION **DPSTraceContext** controls the tracing of data between a PostScript execution context (or contexts) in the Window Server and an application process.

The first argument, *context*, specifies the context to be traced. An application's single context can be returned with **DPSGetCurrentContext**. Applications having more than one execution context can use the constant DPS_ALLCONTEXTS to trace all contexts belonging to them.

The second argument, *flag*, determines whether tracing is enabled. When data tracing is enabled, a copy of the PostScript code generated by an application and the values returned to it by the Window Server is sent to standard error. Values returned to the application are marked by the prepended string:

```
% value returned ==>
```

For applications based on the Application Kit, there are two preferable methods for turning on data tracing: You can use the NXShowPS command-line switch when you launch an application from Terminal. Alternatively, when you run the application under GDB, you can use the **showps** and **shownops** commands to control tracing output.

Only one tracing context can be created for the supplied *context*. If you attempt to create additional tracing contexts for a context that's already being traced, no new context is created and **DPSTraceContext** returns –1.

RETURN **DPSTraceContext** returns 0 if successful in creating a tracing context, or –1 if not.

SEE ALSO **DPSTraceEvents**

DPSTraceEvents

SUMMARY Trace data and events

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSTraceEvents(**DPSContext *context*, int *flag***)**

DESCRIPTION **DPSTraceEvents** controls the tracing of events between a PostScript execution context (or contexts) in the Window Server and an application process.

The first argument, *context*, specifies the context to be traced. An application's single context can be returned with **DPSGetCurrentContext**. Applications having more than one execution context can use the constant DPS_ALLCONTEXTS to trace all contexts belonging to them.

The second argument, *flag*, determines whether tracing is enabled. When event tracing is enabled, information about each event that the application receives is sent to standard error. For example, for a left mouse-down event the listing might look like this:

```
Receiving: LMouseDown at: 343.0,69.0 time: 1271899
   flags: 0x0 win: 6 ctxt: 76128 data: 1111,1
```

The listing displays the fields of the event record: type, location, time, flags, local window number, PostScript execution context, and data. The contents of the data field listing depends on the event type; for instance, in the preceding example the event number and the click count were displayed.

To enable event tracing, you can use the **NXTraceEvents** command-line switch when you launch an application from Terminal. Alternatively, when you run the application under GDB, you can use the **traceevents** and **tracenoevents** commands to control event-tracing output.

SEE ALSO **DPSTraceContext**

DPSUndefineUserObject

SUMMARY Remove a user object

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSUndefineUserObject(**int *index***)**

DESCRIPTION **DPSUndefineUserObject** removes the association between *index* and the PostScript object it refers to, thus destroying the user object. By destroying a user object that's no longer needed, you can let the garbage collector reclaim the previously associated PostScript object.

SEE ALSO **DPSDefineUserObject**

NX_EVENTCODEMASK

SUMMARY Generate an event mask for an event type

DECLARED IN dpsclient/event.h

SYNOPSIS int **NX_EVENTCODEMASK(**int *type***)**

DESCRIPTION This handy utility macro returns an event mask that corresponds to the given (single) event type.

Single-Operator Functions

Note: This section has not been updated and has not received recent technical review. It is included in this release to test the linkage between application development tools and on-line documentation. The information in this section should be considered at best preliminary and subject to change. An updated version of this file will be included in the next release.

PSadjustcursor(float *dx*, float *dy***)**
PSalphaimage(void**)**
PSbasetocurrent(float *bx*, float *by*, float **cx*, float **cy***)**
PSbasetoscreen(float *bx*, float *by*, float **sx*, float **sy***)**
PSbuttondown(boolean **isdown***)**
PScleartrackingrect(int *trectnum*, userobject *gstate***)**
PScomposite(float *srcx*, float *srcy*, float *width*, float *height*, userobject *srcgstate*, float *destx*, float *desty*, int *op***)**

The value passed as *op* should be one of the following:

NX_CLEAR	NX_SIN	NX_SATOP
NX_COPY	NX_DIN	NX_DATOP
NX_SOVER	NX_SOUT	NX_PLUSD
NX_DOVER	NX_DOUT	NX_PLUSL
NX_XOR		

PScompositerect(float *destx*, float *desty*, float *width*, float *height*, int *op***)**

The value passed as *op* should be one *of the constants listed under* **PScomposite**, plus NX_HIGHLIGHT.

PScountframebuffers(int **count***)**
PScountscreenlist(int *context*, int **count***)**
PScountwindowlist(int *context*, int **count***)**
PScurrentactiveapp(int **context***)**
Warning: Don't use this function if you're using the Application Kit.

PScurrentalpha(float **coverage***)**
PScurrentdefaultdepthlimit(int **depth***)**

Warning: Don't use this function if you're using the Application Kit.

PScurrentdeviceinfo(userobject *window*, int **min*, int **max*, boolean **iscolor***)*
PScurrenteventmask(userobject *window*, int **mask***)*
Warning: Don't use this function if you're using the Application Kit.

PScurrentframebuffertransfer(void**)**
PScurrentmouse(userobject *window*, float **x*, float **y***)*
Warning: Don't use this function if you're using the Application Kit.

PScurrentowner(userobject *window*, int **context***)*
PScurrentshowpageprocedure(void)
PScurrentrusage(float **ctime*, float **utime*, float **stime*, int **msgsend*, int **msgrcv*, int **nsignals*, int **nvcsw*, int **nivcsw***)*
PScurrenttobase(float *cx*, float *cy*, float **bx*, float **by***)*
PScurrenttoscreen(float *cx*, float *cx*, float **sx*, float **sy***)*
PScurrentuser(int **uiId*, int **gid***)*
PScurrentwaitcursorenabled(boolean **isenabled***)*
PScurrentwindow(userobject **window***)*
PScurrentwindowalpha(userobject *window*, int **alpha***)*
PScurrentwindowbounds(userobject *window*, float **x*, float **y*, float **width*, float **height***)*
Warning: Don't use this function if you're using the Application Kit.

PScurrentwindowdepth(userobject *window*, int **depth***)*
Warning: Don't use this function if you're using the Application Kit.

PScurrentwindowdepthlimit(userobject *window*, int **depth***)*
Warning: Don't use this function if you're using the Application Kit.

PScurrentwindowdict(userobject *window***)**
Warning: Don't use this function if you're using the Application Kit.

PScurrentwindowlevel(userobject *window*, int **level***)*
PScurrentwriteblock(bool **doesblock***)*
PSdissolve(float *srcx*, float *srcy*, float *sourceWidth*, float *width*, userobject *srcgstate*, float *destx*, float *desty*, float *delta***)**
PSdumpwindow(int *dumplevel*, userobject *window***)**
Warning: Don't use this function if you're using the Application Kit.

PSdumpwindows(int *dumplevel*, userobject *context***)**

Warning: Don't use this function if you're using the Application Kit.

PSfindwindow(float *x*, float *y*, int *place*, userobject *otherwindow*, float **x*', float **y*', userobject **window*, boolean **found***)*

The value passed as place should be one of the following:

NX_ABOVE
NX_BELOW

PSflushgraphics(void**)**
Warning: Don't use this function if you're using the Application Kit.

PSframebuffer(int *index*, int *stringlen*, char *string*[], int **slot*, int **unit*, int **romid*, int **x*, int **y*, int **width*, int **height*, int **maxdepth***)*
PSfrontwindow(int **window***)*
Warning: Don't use this function if you're using the Application Kit.

PShidecursor(void**)**
PShideinstance(float *x*, float *y*, float *width*, float *height***)**
PSmachportdevice(int *width*, int *height*, const int *bbox*[], int *bboxSize*, const float *matrix*[], const char **hostname*, const char **portname*, const char **pixelencoding***)*
PSmovewindow(float *x*, float *y*, userobject *window***)**
Warning: Don't use this function if you're using the Application Kit.

PSnewinstance(void**)**
PSnextrelease(int *size*, char *string*[]**)**
PSobscurecursor(void**)**
PSorderwindow(int *place*, userobject *otherwindow*, int *window***)**
Warning: Don't use this function if you're using the Application Kit.

The value passed as place should be one of the following:

NX_ABOVE
NX_BELOW
NX_OUT

PSosname(int *size*, char *string*[]**)**
PSostype(int **type***)*
PSplacewindow(float *x*, float *y*, float *width*, float *height*, userobject *window***)**†

Warning: Don't use this function if you're using the Application Kit.

PSplaysound(const char **soundname*, int *priority***)**
PSposteventbycontext(int *type*, float *x*, float *y*, int *time*, int *flags*, int *window*, int *subtype*, int *misc0*, int *misc1*, int *context*, boolean **success* **)**
PSreadimage(void**)**
PSrevealcursor(void**)**
PSrightbuttondown(int **isdown***)**
PSrightstilldown(int *eventnum*, boolean **stilldown***)**
PSscreenlist(int *context*, int *count*, int *array*[]**)**
PSscreentobase(float *sx*, float *sy*, float **bx*, float **by***)**
PSscreentocurrent(float *sx*, float *sy*, float **cx*, float **cy***)**
PSsetactiveapp(int *context***)**
Warning: Don't use this function if you're using the Application Kit.

PSsetalpha(float *coverage***)**
PSsetautofill(boolean *flag*, userobject *window***)**
PSsetcursor(float *x*, float *y*, float *mx*, float *my***)**
PSsetdefaultdepthlimit(int *depth***)**
Warning: Don't use this function if you're using the Application Kit.

PSseteventmask(int *mask*, userobject *window***)**
Warning: Don't use this function if you're using the Application Kit.

See the constants listed under "Event Type Masks" in the section "Types and Constants" for a list of *mask* values.

PSsetexposurecolor(void**)**
PSsetflushexposures(boolean *flag***)**
Warning: Don't use this function if you're using the Application Kit.

PSsetframebuffertransfer(void)
PSsetinstance(boolean *flag***)**
PSsetmouse(float *x*, float *y***)**
PSsetowner(userobject *context*, userobject *window***)**
PSsetsendexposed(boolean *flag*, userobject *window***)**†
Warning: Don't use this function if you're using the Application Kit.

PSsetshowpageprocedure(int *window***)**

Warning: Don't use this function if you're using the Application Kit.

PSsettrackingrect(float *x*, float *y*, float *width*, float *height*, boolean *leftbool*, boolean *rightbool*, boolean *insidebool*, int *userdata*, int *trectnum*, userobject *gstate***)**

Note: Only the Form 1 version of **the settrackingrect** operator is offered as a C function.

PSsetwaitcursorenabled(boolean *flag***)**
PSsetwindowdepthlimit(int *depth*, userobject *window***)**
Warning: Don't use this function if you're using the Application Kit.

PSsetwindowdict(userobject *window***)**
Warning: Don't use this function if you're using the Application Kit.

PSsetwindowlevel(int *level*, userobject *window***)**
PSsetwindowtype(int *type*, userobject *window***)**
Warning: Don't use this function if you're using the Application Kit.

PSsetwriteblock(int *flag***)**
PSshow(const char **string***)**
PSshowcursor(void**)**
PSsizeimage(float *x*, float *y*, float *width*, float *height*, int **pixelswide*, int **pixelshigh*, int **bits/sample*, float *matrix*[], boolean **multiproc*, int **ncolors***)**
PSstilldown(int *eventnum*, boolean **stilldown***)**
PStermwindow(userobject *window***)**
Warning: Don't use this function if you're using the Application Kit.

PSwindow(float *x*, float *y*, float *width*, float *height*, int *type*, int **window***)**
Warning: Don't use this function if you're using the Application Kit.

PSwindowdevice(userobject *window***)**
PSwindowdeviceround(userobject *window***)**
PSwindowlist(int *context*, int *count*, int *subarray[]***)**

PostScript Operators

Note: This section has not been updated and has not received recent technical review. It is included in this release to test the linkage between application development tools and on-line documentation. The information in this section should be considered at best preliminary and subject to change. An updated version of this file will be included in the next release.

adjustcursor

SYNOPSIS *dx dy* **adjustcursor** –

Moves the cursor location by (*dx*, *dy*) from its current location. *dx* and *dy* are given in the current coordinate system. If the current device isn't a window, the **invalidid** error is executed.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **currentmouse, setmouse**

alphaimage

SYNOPSIS *pixelswide pixelshigh bits/sample matrix datasrc0 [...datasrcn] multiproc ncolors* **alphaimage** –

Renders an image whose samples include an alpha component. (Most programmers should use **NXImageBitmap()** instead of **alphaimage**.) This operator is similar to the standard **colorimage** operator (as documented by Adobe Systems). However, note the following:

- When supplying the data components, alpha is always given last—either as the last data source (*datasrcn*) if the data is given in separate vectors, or as the last element in a set of interleaved data.
- The *ncolors* operand doesn't account for alpha—the value of *ncolors* is the number of color components only.

ERRORS **invalidid**, **limitcheck**, **rangecheck**, **stackunderflow**, **typecheck**, **undefined**, **undefinedresult**

basetocurrent

SYNOPSIS *bx by* **basetocurrent** *cx cy*

Converts (*bx*, *by*) from the current window's base coordinate system to its current coordinate system. If the current device isn't a window, the **invalidid** error is executed.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **basetoscreen, currenttobase, currenttoscreen, screentobase, screentocurrent**

basetoscreen

SYNOPSIS *bx by* **basetoscreen** *sx sy*

Converts (*bx*, *by*) from the current window's base coordinate system to the screen coordinate system. If the current device isn't a window, the **invalidid** error is executed.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **basetocurrent, currenttobase, currenttoscreen, screentobase, screentocurrent**

buttondown

SYNOPSIS – **buttondown** *isdown*

Returns *true* if the left or only mouse button is currently down; otherwise it returns *false*.

Note: To test whether the mouse button is still down from a mouse-down event, use **stilldown** instead of **buttondown**; **buttondown** will return *true* even if the mouse button has been released and pressed again since the original mouse-down event.

ERRORS none

SEE ALSO **currentmouse, rightbuttondown, rightstilldown, stilldown**

cleartrackingrect

SYNOPSIS *trectnum gstate* **cleartrackingrect** –

Clears the tracking rectangle identified by *trectnum*, as set by **settrackingrect**, in the device referred to by *gstate* (or the current graphics state if *gstate* is **null**). If no such rectangle exists, the **invalidid** error is executed.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **settrackingrect**

composite

SYNOPSIS *srcx srcy width height srcgstate destx desty op* **composite** –

Performs the compositing operation specified by *op* between pairs of pixels in two images, a source and a destination. The source pixels are in the window device referred to by the *srcgstate* graphics state, and the destination pixels are in the current window. If *srcgstate* is **null**, the current graphics state is assumed. If either graphics state doesn't refer to a window device, the **invalidid** error is executed.

The rectangle specified by *srcx*, *srcy*, *width*, and *height* defines the source image. The outline of the rectangle may cross pixel boundaries due to fractional coordinates, scaling, or rotated axes. The pixels included in the source are all those that the outline of the rectangle encloses or enters.

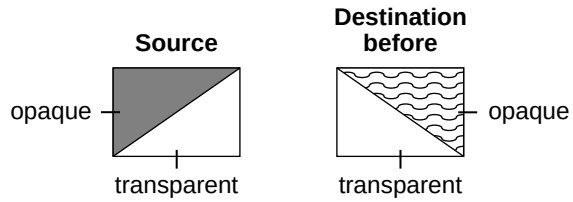
The destination image has the same size, shape, and orientation as the source; *destx* and *desty* give destination's location image compared to the source. (Even if the two graphic states have different orientations, the images will not; **composite** will not rotate images.)

Both images are clipped to the frame rectangles of their respective windows. The destination image is further clipped to the clipping path of the current graphics state. The result of a composite operation replaces the destination image.

op specifies the compositing operation. The choices for *op* and the result of each operation are given in the following illustration.

ERRORS **invalidid**, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **compositerect, setalpha**



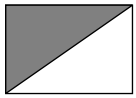
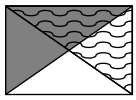
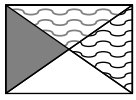
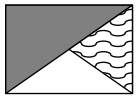
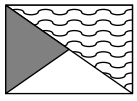
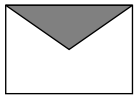
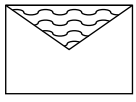
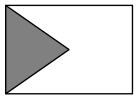
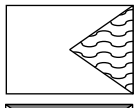
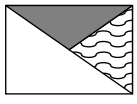
Operation	Destination after	
Copy		Source image.
Clear		Transparent.
PlusD		Sum of source and destination images, with color values approaching 0 as a limit.
PlusL		Sum of source and destination images, with color values approaching 1 as a limit. (PlusL is not implemented for the MegaPixel Display.)
Sover		Source image wherever source image is opaque, and destination image elsewhere.
Dover		Destination image wherever destination image is opaque, and source image elsewhere.
Sin		Source image wherever both images are opaque, and transparent elsewhere.
Din		Destination image wherever both images are opaque, and transparent elsewhere.
Sout		Source image wherever source image is opaque but destination image is transparent, and transparent elsewhere.
Dout		Destination image wherever destination image is opaque but source image is transparent and transparent elsewhere.
Satop		Source image wherever both images are opaque, destination image wherever destination image is opaque but source image is transparent, and transparent elsewhere.

Figure 0-1. Compositing Operations

compositerect

SYNOPSIS *destx desty width height op* **compositerect** –

In general, this operator is the same as the **composite** operator except that there's no real source image. The destination is in the current graphics state; *destx*, *desty*, *width*, and *height* describe the destination image in that graphics state's current coordinate system. The effect on the destination is as if there were a source image filled with the color and coverage specified by the graphics state's current color parameter. *op* has the same meaning as the *op* operand of the **composite** operator; however, one additional operation, Highlight, is allowed.

On the MegaPixel Display, Highlight turns every white pixel in the destination rectangle to light gray and every light gray pixel to white, regardless of the pixel's coverage value. Repeating the same operation reverses the effect. (Highlight may act differently on other devices. For example, on displays that assign just one bit per pixel, it would invert every pixel.)

Note: The Highlight operation doesn't change the value of a pixel's coverage component. To ensure that the pixel's color and coverage combination remains valid, Highlight operations should be temporary and should be reversed before any further compositing.

For **compositerect**, the pixels included in the destination are those that the outline of the specified rectangle encloses or enters. The destination image is clipped to the frame rectangle and clipping path of the window in the current graphics state.

If the current graphics state doesn't refer to a window device, the **invalidid** error is executed.

ERRORS **invalidid**, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **composite, setalpha**

copypage

Warning: This standard PostScript operator has no effect in the OPENSTEP implementation of the Display PostScript system.

countframebuffers

SYNOPSIS – **countframebuffers** *count*

Returns the number of frame buffers that the Window Server is actually using.

ERRORS **stackoverflow**

SEE ALSO **framebuffer**

countscreenlist

SYNOPSIS *context* **countscreenlist** *count*

Returns the number of windows in the screen list that were created by the PostScript context specified by *context*. This is in contrast with **countwindowlist**, which returns the number of windows created by the context without regard to their inclusion in the screen list.

If *context* is 0, all windows in the screen list are counted, without regard to the context that created them.

ERRORS **invalidid**, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **countwindowlist, screenlist, windowlist**

countwindowlist

SYNOPSIS *context* **countwindowlist** *count*

Returns the number of windows that were created by the PostScript context specified by *context*. This is in contrast with **countscreenlist**, which returns the number of windows in the screen list that were created by the PostScript context specified by *context*.

If *context* is 0, all windows are counted, without regard to the context that created them.

ERRORS **stackunderflow**, **typecheck**

SEE ALSO **countscreenlist, screenlist, windowlist**

currentactiveapp

SYNOPSIS – **currentactiveapp** *context*

Warning: Don't use this operator if you're using the Application Kit.

Returns the active application's context. This operator is used by the window packages to assist with wait cursor operation.

ERRORS **stackoverflow**

SEE ALSO **setactiveapp**

currentalpha

SYNOPSIS – **currentalpha** *coverage*

Returns the coverage parameter of the current graphics state.

ERRORS none

SEE ALSO **composite, setalpha**

currentdefaultdepthlimit

SYNOPSIS – **currentdefaultdepthlimit** *depth*

Warning: Don't use this operator if you're using the Application Kit. Use Window's **defaultDepthLimit** class method instead.

Returns the current context's default depth limit. This value determines a new window's depth limit.

ERRORS **stackoverflow**

SEE ALSO **setdefaultdepthlimit**, **setwindowdepthlimit**, **currentwindowdepthlimit**, **currentwindowdepth**

currentdeviceinfo

SYNOPSIS *window* **currentdeviceinfo** *min max iscolor*

Returns device-related information about the current state of *window*. *min* and *max* are the smallest and largest number of bits per sample, respectively, and *iscolor* is a boolean value indicating whether the device is a color device.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

currenteventmask

SYNOPSIS *window* **currenteventmask** *mask*

Warning: Don't use this operator if you're using the Application Kit. Use Window's **eventMask** method instead.

Returns the current Window Server-level event mask for the specified window.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **seteventmask**

currentframebuffertransfer

SYNOPSIS *fbnum* **currentframebuffertransfer** *redproc greenproc blueproc grayproc*

Returns the current transfer functions in effect for the framebuffer indexed by *fbnum*. *fbnum* ranges from 0 to (countframebuffers – 1).

ERRORS **invalidid, stackunderflow, typecheck**

SEE ALSO **setframebuffertransfer, countframebuffers, framebuffer**

currentmouse

SYNOPSIS *window* **currentmouse** *x y*

Warning: Don't use this operator if you're using the Application Kit. Use Window's **getMouseLocation:** instead.

Returns the current x and y coordinates of the mouse location in the base coordinate system of the specified window. If the mouse isn't inside the specified window, these coordinates may be outside the coordinate range defined for the window. If *window* is 0, the current mouse position is returned relative to the screen coordinate system.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **basetocurrent**, **basetoscreen**, **buttondown**, **rightbuttondown**, **rightstilldown**, **setmouse**, **stilldown**

currentowner

SYNOPSIS *window* **currentowner** *context*

Returns a number identifying the PostScript context that currently owns the specified window. By default, this is the PostScript context that created the window.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **setowner**, **termwindow**, **window**

currentshowpageprocedure

SYNOPSIS *window* **currentshowpageprocedure** *proc*

Returns the PostScript procedure that's executed when the **showpage** operator is executed while the specified window is the current device.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **setshowpageprocedure**

currentrusage

SYNOPSIS – **currentrusage** *ctime utime stime msgsend msgrcv nsignals nvcsw nivcsw*

Returns information about the current time of day and about resource usage by the Window Server, as provided by the system call **getrusage()**. The items returned, and their types, are as follows:

Name	Type	Value
ctime	float	Current time in seconds, modulo 10000
utime	float	User time for the Server process in seconds
stime	float	System time for the Server process in seconds
msgsend	int	Messages sent by the Server to clients
msgrcv	int	Message received by the Server from clients
nsignals	int	Number of signals received by the Server process
nvcsw	int	Number of voluntary context switches

Name	Type	Value
nivcsw	int	Number of involuntary context switches

currenttobase

SYNOPSIS *cx cy* **currenttobase** *bx by*

Converts (*cx*,*cy*) from the current coordinate system of the current window to its base coordinate system. If the current device isn't a window, the **invalidid** error is executed.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **basetocurrent**, **basetoscreen**, **currenttoscreen**, **screentobase**, **screentocurrent**

currenttoscreen

SYNOPSIS *cx cy* **currenttoscreen** *sx sy*

Converts (*cx*, *cy*) from the current coordinate system of the current window to the screen coordinate system. If the current device isn't a window, the **invalidid** error is executed.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **basetocurrent**, **basetoscreen**, **currenttobase**, **screentobase**, **screentocurrent**

currentuser

SYNOPSIS – **currentuser** *uid gid*

Returns the user id (*uid*) and the group id (*gid*) of the user currently logged in on the console of the machine that's running the Window Server.

ERRORS **stackoverflow**

currentwaitcursorenabled

SYNOPSIS *context* **currentwaitcursorenabled** *isenabled*

Returns the state of *context*'s wait cursor flag. If *context* is 0, returns the state of the global wait cursor flag.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **setwaitcursorenabled**

currentwindow

SYNOPSIS – **currentwindow** *window*

Returns the window number of the current window. Executes the **invalidid** error if the current device isn't a window.

ERRORS **invalidid**

SEE ALSO **windowdeviceround**

currentwindowalpha

SYNOPSIS *window* **currentwindowalpha** *alpha*

Returns an integer indicating whether the Window Server is currently storing alpha values for the specified window. Possible *alpha* values are:

–2	Window is opaque; alpha values are explicitly allocated.

0	Alpha values are stored explicitly
2	Window is opaque; no explicit alpha

ERRORS **invalidid**, **stackunderflow**, **typecheck**

currentwindowbounds

SYNOPSIS *window* **currentwindowbounds** *x y width height*

Warning: Don't use this operator if you're using the Application Kit. Use Window's **getFrame:** or Application's **getScreenSize:** method instead.

Returns the location and size of the window in screen coordinates. Pass 0 for *window* to get the size of the entire workspace (the smallest rectangle that encloses all active screens).

x and *y* will be in the range [–215, 215 –1]; *width* and *height* will be in the range [0, 10000].

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **movewindow**, **placewindow**

currentwindowdepth

SYNOPSIS *window* **currentwindowdepth** *depth*

Warning: Don't use this operator if you're using the Application Kit.

Returns *window*'s current depth. The **invalidid** error is executed if *window* doesn't exist.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **setwindowdepthlimit**, **currentwindowdepthlimit**, **setdefaultdepthlimit**, **currentdefaultdepthlimit**

currentwindowdepthlimit

SYNOPSIS *window* **currentwindowdepthlimit** *depth*

Warning: Don't use this operator if you're using the Application Kit. Use Window's **depthLimit** method instead.

Returns the window's current depth limit, the maximum depth to which the window can be promoted. Unless altered by the **setwindowdepthlimit** operator, a window's depth limit is equal to its context's default depth limit. The **invalidid** error is executed if *window* doesn't exist.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **setwindowdepthlimit**, **currentwindowdepth**, **setdefaultdepthlimit**, **currentdefaultdepthlimit**

currentwindowdict

SYNOPSIS *window* **currentwindowdict** *dict*

Warning: Don't use this operator if you're using the Application Kit.

Returns the specified window's dictionary.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **setwindowdict**

currentwindowlevel

SYNOPSIS *window* **currentwindowlevel** *level*

Returns *window*'s tier. Executes the **invalidid** error if *window* doesn't exist.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **setwindowlevel**

currentwriteblock

SYNOPSIS – **currentwriteblock** *doesblock*

Returns whether the Window Server delays sending data to a client application whenever the Server's output buffer fills. **currentwriteblock** assumes the current context. If *doesblock* is *true*, the Server waits until the buffer can accept more data. If *doesblock* is *false*, the Server discards data that can't be accepted immediately.

ERRORS **none**

SEE ALSO **setwriteblock**

dissolve

SYNOPSIS *srcx srcy width height srcgstate destx desty delta* **dissolve** –

The effect of this operation is a blending of a source and a destination image. The first seven arguments choose source and destination pixels as they do for **composite**. The exact fraction of the blend is specified by *delta*, which is a floating-point number between 0.0 and 1.0; the resulting image is:

delta *source + (1– *delta*)*destination

If *srcgstate* is null, the current graphics state is assumed. If *srcgstate* or the current graphics state does not refer to a window device, this operator executes the **invalidid** error.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **composite**

dumpwindow

SYNOPSIS *dumplevel window* **dumpwindow** –

Warning: Don't use this operator if you're using the Application Kit.

Prints information about *window* to the standard output file. Only *dumplevel* 0 is implemented. The information printed is the position and number of bytes of backing storage for the window.

ERRORS **invalidid**, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **dumpwindows**

dumpwindows

SYNOPSIS *dumplevel context* **dumpwindows** –

Warning: Don't use this operator if you're using the Application Kit.

Prints information about all windows owned by *context* to the standard output file. If *context* is 0, it prints information about all windows. Only *dumplevel* 0 is implemented.

ERRORS **invalidid**, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **dumpwindow**

erasepage

SYNOPSIS – **erasepage** –

Warning: This standard operator is different in the OPENSTEP implementation.

Erases the entire window to opaque white.

ERRORS **invalidid**

SEE ALSO **copypage**, **showpage**

findwindow

SYNOPSIS *x y place otherwindow* **findwindow** *x' y' window found*

findwindow starts from a given position in the screen list, as explained below, and searches for the first window below that position that contains the point (*x*, *y*). The *x* and *y* values are given in screen coordinates.

The starting position is determined by *place* and *otherwindow*. *place* can be **Above** or **Below**, and *otherwindow* is the window number of a window in the screen list. If you specify **Above 0**, **findwindow** checks all windows in the screen list.

If a window containing the point is found, **findwindow** returns *true*, along with the window number and the corresponding location in the base coordinate system of the window. Otherwise, it returns *false*, and the values of *x'*, *y'*, and *window* are undefined.

ERRORS **rangecheck**, **stackunderflow**, **typecheck**

flushgraphics

SYNOPSIS – **flushgraphics** –

Warning: Don't use this operator if you're using the Application Kit. Use Window's **flushWindow** method instead.

Flushes to the screen all drawing done in the current buffered window. If the current window is retained or nonretained, **flushgraphics** has no effect.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

framebuffer

SYNOPSIS *index string* **framebuffer** *name slot unit romid x y width height maxdepth*

Provides information on the active frame buffer specified by *index*, where *index* ranges from 0 to **countframebuffers**–1. *string* must be large enough to hold the resulting name of the frame buffer. *slot* is the NeXTbus™ slot the frame buffer is physically occupying. If a board supports multiple frame buffers, *unit* uniquely identifies the frame buffer within a slot. The ROM product code is returned in *romid*. The bottom left corner of the frame buffer is returned in *x* and *y* (relative to the screen coordinate system). The size of the frame buffer in pixels is returned in *width* and *height*. *maxdepth* is the maximum depth displayable on this frame buffer (for example, NSTwentyFourBitRGBDepth).

The **limitcheck** error is executed if *string* isn't large enough to hold *name*. The **rangecheck** error is executed if *index* is out of bounds.

ERRORS **limitcheck**, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **countframebuffers**

frontwindow

SYNOPSIS – **frontwindow** *window*

Warning: Don't use this operator if you're using the Application Kit.

Returns the window number of the frontmost window on the screen. If there aren't any windows on the screen, **frontwindow** returns 0.

ERRORS none

SEE ALSO **orderwindow**

hidecursor

SYNOPSIS – **hidecursor** –

Removes the cursor from the screen. It remains in effect until balanced by a call to **showcursor**.

ERRORS none

SEE ALSO **obscurecursor**, **showcursor**

hideinstance

SYNOPSIS *x y width height* **hideinstance** –

In the current window, **hideinstance** removes any instance drawing from the rectangle specified by *x*, *y*, *width*, and *height*. *x*, *y*, *width*, and *height* are given in the window's current coordinate system.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **newinstance**, **setinstance**

image

SYNOPSIS *dict* **image** –

Allows a window's graphics state object to be used as a source of sample data. dict must be an image dictionary in which only those keys listed in the following table are significant:

Key	Type	Value or Meaning
ImageType	integer	(*Required*) Must be 2.

Key	Type	Value or Meaning
XOrigin	real	(*Required*) X origin of the source rectangle in user space coordinates as specified by the transformation in the DataSource entry.
YOrigin	real	(Required) Y origin of the same.
Width	real	(Required) Width of the same.
Height	real	(Required) Height of the same.
ImageMatrix	array	(Required) The transformation matrix.
DataSource	gstate	(Required) A graphics state object that contains the device that will be used as the source of sample data. This device will also be used to determine the pixel representation for the source, and the color space to be used by the image.
Interpolate	boolean	(Optional) Request for image interpolation.
UnpaintedPath	(various)	(Return value) If some of the pixels in the source weren't available (because of clipping), then the UnpaintedPath entry contains a userpath in the current (destination) user space that encloses the area that couldn't be filled.
PixelCopy	boolean	(Optional) If true, indicates that the source pixels should be copied directly, without going through the normal color conversion, transfer function, or halftoning. The bits per pixel of the source must match the bits per pixel of the destination, otherwise a typecheck error will occur. If false or not present, the pixels will be imaged in the usual way.

ERRORS **invalidid, rangecheck, stackunderflow, typecheck**

SEE ALSO **alphaimage**

initgraphics

SYNOPSIS – **initgraphics** –

Warning: This standard operator has additional effects in the OPENSTEP implementation of the Display PostScript system.

In addition to the effects documented by Adobe, this operator sets the coverage parameter in the current window's graphics state to 1 (opaque) and turns off instance drawing

ERRORS none

SEE ALSO **hideinstance**, **newinstance**, **setalpha**, **setinstance**

machportdevice

SYNOPSIS *width height bbox matrix hostname portname pixelencoding* **machportdevice** –

Sets up a PostScript device that can provide a generic rendering service for device-control programs requiring page bitmaps from PostScript documents. For each rendered page, **machportdevice** sends a Mach message containing the page bitmap to a port that has been registered with the name server on the network.

width and *height* are integers that determine the number of device pixels for the page.

bbox is an array of integers that defines the rectangle (by giving its lower left and upper right corners) that encompasses the imageable area. The array takes the form

[*lowerLeftX lowerLeftY upperRightX upperRightY***]**

For the common case where the entire raster is imageable, *bbox* may be expressed as an empty array. **If** *bbox* isn't in the correct form, or if any portion of the rectangle it expresses falls outside [0 0 *width height*], a **rangecheck** results. The bitmap data is stored in x-axis major indexing order. The device coordinate of the lower left corner of the first pixel is (0,0), the coordinate of the next pixel is (1,0) and so on for the entire scanline. Scanlines are long-word aligned.

matrix is the default transformation matrix for the device.

hostname and *portname* are strings that together identify the port that will receive the Mach messages. If *hostname* is empty, the local host is assumed. If it's "*", the port is searched for on all available hosts. If (in any case) the port can't be found, a **rangecheck** results.

pixelencoding is a dictionary describing the format for the image data rendered by the Window Server. It should contain these entries:

Key	Type	Value
samplesPerPixel	integer	Must be 1.
bitsPerSample	integer	Must be 1 or 2.
colorSpace	integer	Color space specification (see below).
isPlanar	boolean	*true* if sample values are stored in separate arrays (currently must be *false*).
defaultHalftone	dictionary	Passed to **sethalftone** during device creation to set up device default halftone.
initialTransfer	procedure	Passed to **settransfer** during device creation to set up the initial transfer function for device.
jobTag	integer	Allows **machportdevice** to tag rendering jobs. This value is included in the **jobTag** field of all printpage messages generated by this device.

The value of **colorSpace** should be one of the following values, predefined in **nextdict**:

Name	Value	Description
NSOneIsBlackColorSpaceNumber	0	Monochromatic, high sample value is black.
NSOneIsWhiteColorSpaceNumber	1	Monochromatic, high sample value is white.
NSRGBColorSpaceNumber	2	RGB, (1,1,1) is white.
NSCMYKColorSpaceNumber	5	CMYK, (0,0,0,0) is white.

Only the following combinations of **colorSpace** and **bitsPerSample** are supported:

colorSpace	bitsPerSample
NSOneIsBlackColorSpaceNumber	1
NSOneIsWhiteColorSpaceNumber	2

These read-only pixel-encoding dictionaries are predefined in **nextdict**:

Name	Description
NeXTLaser-300	NeXT Laser Printer at 300 dpi resolution
NeXTLaser-400	NeXT Laser Printer at 400 dpi resolution
NeXTMegaPixelDisplay	MegaPixel Display's 2 bits-per-pixel gray

The pagebuffer data is passed out-of-line, appearing in the receiving application's address space. (If the receiver is on the same host, the received pagebuffer references the same physical memory as the Window Server's pagebuffer, and is mapped copy-on-write.) The application should use **vm_deallocate()** to release the pagebuffer memory when it's no longer needed. The receiver must acknowledge receipt of the data by sending a simple **msg_header_t** (with **msg_id** == NX_PRINTPAGEMSGID) back to the **remote_port** passed in the print message. The Window Server will not continue executing the page description until acknowledgement is received.

If more than one copy of the page is needed (through either the **copypage** or **#copies** mechanism) each copy is sent as a separate message. In this case the same pagebuffer will be sent in multiple messages. The **letter**, **legal**, and **note** page types are gracefully ignored.

Messaging errors cause the **invalidaccess** error to be executed.

EXAMPLES This example sets up a 400 dpi 8.5 by 11 inch page on a raster with upper left origin (as with the NeXT 400 dpi Laser Printer) and sends its print page messages to the port named "nlp-123" on the local host:

```
/dpi 400 def
/width dpi 8.5 mul cvi def
/height dpi 11 mul cvi def

width height      % page bitmap dimensions in pixels
[]                % use it all
[dpi 72 div 0 0 dpi -72 div 0 height] % device transform
() (nlp-123)      % host (local) & port
NeXTLaser-400     % pixel-encoding description
machportdevice
```

This example sets up an 8 by 10 inch page on the same 8.5 by 11 inch page. It specifies a 400 dpi raster with 1/4 inch horizontal margins and 1/2 inch vertical margins:

```
/dpi 400 def
/width dpi 8.5 mul cvi def
/height dpi 11 mul cvi def
/topdots dpi .5 mul cvi def
/leftdots dpi .25 mul cvi def

width height         % page bitmap dimensions in pixels
[
   leftdots
   topdots
   width leftdots sub
   height topdots sub
]                    % imageable area of bounding box
[
   dpi 72 div
   0
   0
   dpi -72 div
   leftdots
   height topdots sub
]                    % device transform
() (nlp-123)         % host (local) & port
NeXTLaser-400        % pixel-encoding description
machportdevice
```

Note that in this example, the user space origin is at the lower left corner of the imageable area (*leftdots*, *height-topdots*) in the device raster coordinate system. Usually, the imageable area is meant to correspond with the ultimate destination of the bits. For example, a printer may have a constant-sized pagebuffer with a fixed orientation in the paper path, but be able to accept various sizes of paper. In this case, the page bitmap size will always be fixed, but the imageable area and default device transformation can be adjusted to make the user space origin appear at the lower left corner of each printed page.

ERRORS **invalidaccess**, **limitcheck**, **rangecheck**, **stackunderflow**, **typecheck**

movewindow

SYNOPSIS *x y window* **movewindow** –

Warning: Don't use this operator if you're using the Application Kit. Use Window's **moveTo::** method instead.

Moves the lower left corner of the specified window to the screen coordinates (*x*, *y*). No portion of the repositioned window can have an x or y coordinate with an absolute value greater than 16000. The operands can be integer, real, or radix numbers; however, they are converted to integers in the Window Server by rounding toward 0.

The window need not be the frontmost window. This operator doesn't change *window*'s ordering in the screen list.

ERRORS **invalidid**, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **currentwindowbounds**, **placewindow**

newinstance

SYNOPSIS – **newinstance** –

Removes any instance drawing from the current window.

ERRORS **invalidid**

SEE ALSO **hideinstance**, **setinstance**

nextrelease

SYNOPSIS – **nextrelease** *string*

Returns version information about this release.

ERRORS **stackoverflow**

SEE ALSO **osname**, **ostype**

NextStepEncoding

SYNOPSIS – **NextStepEncoding** *array*

Pushes the NextStepEncoding vector on the operand stack. This is a 256-element array, indexed by character codes, whose values are the character names for those codes.

ERRORS **stackoverflow**

obscurecursor

SYNOPSIS – **obscurecursor** –

Removes the cursor image from the screen until the next time the mouse is moved. It's usually called in response to typing by the user, so the cursor won't be in the way. If the cursor has already been removed due to an **obscurecursor** call, **obscurecursor** has no effect.

ERRORS none

SEE ALSO **hidecursor**, **revealcursor**

orderwindow

SYNOPSIS *place otherwindow window* **orderwindow** –

Warning: Don't use this operator if you're using the Application Kit. Use Window's **orderWindow:relativeTo:** instead.

Orders *window* in the screen list as indicated by *place* and *otherwindow*. *place* can be **Above**, **Below**, or **Out**:

- If *place* is **Above** or **Below**, the window is placed in the screen list immediately above or below the window specified by *otherwindow*.
- If *place* is **Above** or **Below** and *otherwindow* is 0, the window is placed above or below all windows in its tier.
- If *place* is **Above** or **Below**, *otherwindow* must be a window in the screen list; otherwise, the **invalidid** error is executed.
- If *place* is **Out**, *otherwindow* is ignored, and the window is removed from the screen list, so it won't appear anywhere on the screen. Windows that aren't in the screen list don't receive user events.

Note: **orderwindow** doesn't change which window is the current window.

ERRORS **invalidid**, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **frontwindow**

osname

SYNOPSIS – **osname** *string*

Returns a string identifying the operating system of the Window Server's current operating environment. **osname** is defined in the **statusdict** dictionary, a dictionary that defines operators specific to a particular implementation of the PostScript language. **osname** can be executed as follows:

```
statusdict /osname get exec
```

ERRORS none

SEE ALSO **nextrelease**, **ostype**

ostype

SYNOPSIS – **ostype** *int*

Returns a number identifying the operating system of the Window Server's current operating environment. **ostype** is defined in the **statusdict** dictionary, a dictionary that defines operators specific to a particular implementation of the PostScript language. **ostype** can be executed as follows:

```
statusdict /ostype get exec
```

ERRORS none

SEE ALSO **nextrelease**, **osname**

placewindow

SYNOPSIS *x y width height window* **placewindow** –

Warning: Don't use this operator if you're using the Application Kit. Use Window's **placeWindow:** method instead.

Repositions and resizes the specified window, effectively allowing it to be resized from any corner or point. *x*, *y*, *width*, and *height* are given in the screen coordinate system. No portion of the repositioned window can have an x or y coordinate with an absolute value greater than 16000; *width* and *height* must be in the range from 0 to 10000. The four operands can be integer or real numbers; however, they are converted to integers in the Window Server by rounding toward 0.

placewindow places the lower left corner of the window at (*x*, *y*) and resizes it to have a width of *width* and a height of *height*. The pixels that are in the intersection of the old and new positions of the window survive unchanged (see Figure 0-2). Any other areas of the newly positioned window are filled with the window's exposure color (see **setexposurecolor**).

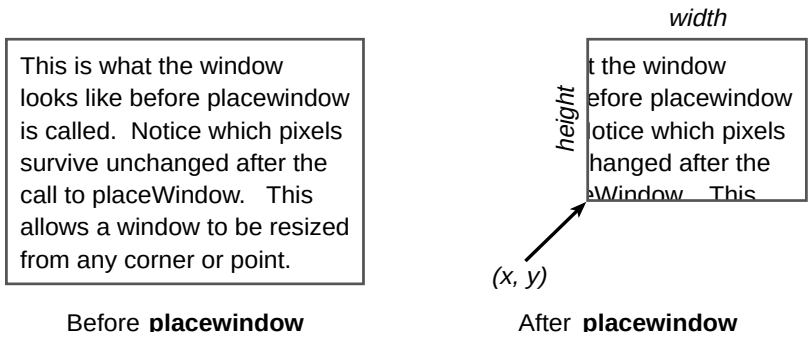


Figure 0-2. **placewindow**

After moving or resizing a window with **placewindow**, you must execute the **initmatrix** and **initclip** operators to reestablish the window's default transformation matrix and default clipping path.

ERRORS **invalidid**, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **currentwindowbounds**, **movewindow**, **setexposurecolor**

playsound

SYNOPSIS *soundname priority* **playsound** –

Plays the sound *soundname*. The Window Server searches for a standard soundfile of the name

*soundname***.snd**

The search progresses through the following directories in the order given, stopping when the sound is located.

~/Library/Sounds
/LocalLibrary/Sounds
/NextLibrary/Sounds

No error occurs if the soundfile isn't found: The operator has no effect.

The soundfile's playback is assigned the priority level *priority*. The playback interrupts any currently playing sound of the same or lower priority level.

ERRORS **stackunderflow**, **typecheck**

posteventbycontext

SYNOPSIS *type x y time flags window subtype misc0 misc1 context* **posteventbycontext** *success*

Posts an event to the specified context. The nine parameters preceding the context parameter coincide with the NXEvent structure members (see "Types and Constants" for the definition of the NXEvent structure). The *x* and *y* coordinate arguments are passed directly to the receiving context without undergoing any transformations. *window* is the Window Server's global window number. Returns *true* if the event was successfully posted to *context*, and *false* otherwise.

You use this operator to post an application-defined event to your own application. Use Mach messaging to communicate between applications.

ERRORS **stackunderflow**, **typecheck**

readimage

SYNOPSIS *x y width height proc0 [... procn-1] string bool* **readimage** –

Reads the pixels that make up the rectangular image described by *x*, *y*, *width*, and *height* in the current window. (Most programmers should use **NXReadBitmap()** instead of this operator.)

Usually the image is the rectangle that has a lower left corner of (*x*, *y*) in the current coordinate system and a width and height of *width* and *height*. If the axes have been rotated so that the sides of the rectangle are no longer aligned with the edges of the screen, the image is the smallest screen-aligned rectangle enclosing the given rectangle.

You typically call **sizeimage** before **readimage** (sending it the same *x*, *y*, *width*, and *height* values you'll use for **readimage**) to find out *ncolors*, the number of color components that **readimage** must read. *bool* is a boolean value that determines whether **readimage** reads the alpha component in addition to the color component(s) for each pixel. The total number of components to be read for each pixel, together with the *multiproc* value returned by **sizeimage**, determine *n*, the number of procedures that **readimage** requires. If *multiproc* is *false*, *n* equals 1. Otherwise, *n* equals the number of color components plus the alpha component, if present.

readimage executes the procedures in order, 0 through *n–1*, as many times as needed. For each execution, it pushes on the operand stack a substring of *string* containing the data from as many scanlines as possible. The length of the substring is a multiple of

width * *bits/sample* * (*samples/proc*) / 8

rounded up to the nearest integer. (The *width* and *bits/sample* values are provided by the **sizeimage** operator. *samples* is the number of color components plus 1 for the alpha component, if present.)

The samples are ordered and packed as they are for the **image**, **colorimage**, or **alphaimage** operator. For example, the alpha component is last and, if necessary, extra bits fill up the last character of every scanline. Note that the contents of *string* are valid only for the duration of one call to one procedure because the same string is reused on each procedure call. The **rangecheck** error is executed if *string* isn't long enough for one scanline.

ERRORS **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **alphaimage**, **sizeimage**

revealcursor

SYNOPSIS – **revealcursor** –

Redisplays the cursor that was hidden by a call to **obscurecursor**, assuming that the cursor hasn't already been revealed by mouse movement. If the cursor hasn't been removed from the screen by a call to **obscurecursor**, **revealcursor** has no effect.

ERRORS none

SEE ALSO **obscurecursor**

rightbuttondown

SYNOPSIS – **rightbuttondown** *isdown*

Returns *true* if the right mouse button is currently down; otherwise it returns *false*.

Note: To test whether the right mouse button is still down from a mouse-down event, use **rightstilldown** instead of **rightbuttondown**; **rightbuttondown** will return *true* even if the mouse button has been released and pressed again since the original mouse-down event.

ERRORS none

SEE ALSO **buttondown**, **currentmouse**, **rightstilldown**, **stilldown**

rightstilldown

SYNOPSIS *eventnum* **rightstilldown** *stilldown*

Returns *true* if the right mouse button is still down from the mouse-down event specified by *eventnum*; otherwise it returns *false*. *eventnum* should be the number stored in the **data** component of the event record for an event of type **Rmousedown**.

ERRORS **stackunderflow**, **typecheck**

SEE ALSO **buttondown**, **currentmouse**, **rightbuttondown**, **stilldown**

screenlist

SYNOPSIS *array context* **screenlist** *subarray*

Fills the array with the window numbers of all windows in the screen list that are owned by the PostScript context specified by *context*. It returns the subarray containing those window numbers, in order from front to back. If *array* isn't large enough to hold them all, this operator will return the frontmost windows that fit in the array.

If *context* is 0, all windows in the screen list are returned.

EXAMPLE `This example yields an array containing the window numbers of all windows in the screen list that are owned by the current PostScript context:`

```
currentcontext
countscreenlist             % find out how many windows
array                       % create array to hold them
currentcontext screenlist   % fill it in
```

ERRORS **invalidaccess**, **invalidid**, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **countscreenlist**, **countwindowlist**, **windowlist**

screentobase

SYNOPSIS *sx sy* **screentobase** *bx by*

Converts (*sx*, *sy*) from the screen coordinate system to the current window's base coordinate system. If the current device isn't a window, the **invalidid** error is executed.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **basetocurrent**, **basetoscreen**, **currenttobase**, **currenttoscreen**, **screentocurrent**

screentocurrent

SYNOPSIS *sx sy* **screentocurrent** *cx* cy

Converts (*sx*,*sy*) from the screen coordinate system to the current coordinate system of the current window. If the current device isn't a window, the **invalidid** error is executed.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **basetocurrent**, **basetoscreen**, **currenttobase**, **currenttoscreen**, **screentobase**

setactiveapp

SYNOPSIS *context* **setactiveapp** –

Warning: Don't use this operator if you're using the Application Kit.

Records the active application's main (usually only) context. **setactiveapp** is used by the window packages to assist with wait cursor operation.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **currentactiveapp**

setalpha

SYNOPSIS *coverage* **setalpha** –

Sets the coverage parameter in the current window's graphics state to *coverage*. *coverage* must be a number between 0 and 1, with 0 corresponding to transparent, 1 corresponding to opaque, and intermediate values corresponding to partial coverage. This establishes how much background shows through for purposes of compositing.

ERRORS **stackunderflow**, **typecheck**, **undefined**

SEE ALSO **composite**, **currentalpha**

setautofill

SYNOPSIS *flag window* **setautofill** –

Applies only to nonretained windows; sets the autofill property of *window* to *the value of flag*. If *true*, newly exposed areas of the window or areas created by **placewindow** will automatically be filled with the window's exposure color. If *false*, these areas will not change (typically they will continue to contain the image of the last window in that area). If the current device is not a window, this operator executes the **invalidid** error.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **placewindow**, **setexposurecolor**, **setsendexposed**

setcursor

SYNOPSIS *x y mx my* **setcursor** –

Sets the cursor image and hot spot. Rather than executing this operator directly, you'd normally use a NXCursor object to define and manage cursors.

A cursor image is derived from a 16-pixel-square image in a window that's generally placed off-screen. The *x* and *y* operands specify the upper left corner of the image in the window's current coordinate system. The *mx* and *my* operands specify the relative offset (in units of the current coordinate system) from (*x*, *y*) to the *hot spot*, the point in the cursor that coincides with the mouse location. Assuming the current coordinate system is the base coordinate system, *mx* must be an integer from 0 to 16, and *my* must be an integer from 0 to –16. After **setcursor** is executed, the image in the window is no longer needed.

The cursor is placed on the screen using Sover compositing. The cursor's opaque areas (alpha = 1) completely cover the background, while its transparent areas (alpha < 1) allow the background to show through to a greater extent depending on the alpha values present in the cursor image.

Note: To make the off-screen window transparent, you can use **compositerect** with **Clear**.

The **rangecheck** error is executed if the image doesn't lie entirely within the specified window or if the point (*mx*, *my*) isn't inside the image. If the current device isn't a window, the **invalidid** error is executed.

ERRORS **invalidid**, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **hidecursor**, **obscurecursor**, **setmouse**

setdefaultdepthlimit

SYNOPSIS *depth* **setdefaultdepthlimit** –

Warning: Don't use this operator if you're using the Application Kit.

Sets the current context's default depth limit to *depth*. The Window Server assigns each new context a default depth limit equal to the maximum depth supported by the system. When a new window is created, its depth limit is set to its context's default depth limit.

These depths are defined in **nextdict**:

Depth	Meaning
NSTwoBitGrayDepth	1 spp, 2bps, 2bpp, planar
NSEightBitGrayDepth	1 spp, 8bps, 8bpp, planar
NSTwelveBitRGBDepth	3 spp, 4bps, 16bpp, interleaved
NSTwentyFourBitRGBDepth	3 spp, 8bps, 32bpp, interleaved

where *spp* is the number of samples per pixel; *bps* is the number of bits per sample; and *bpp* is the number of bits per pixel, also known as the window's depth. (The samples-per-pixel value excludes the alpha sample, if present.) *planar* and *interleaved* refer to how the sample data is configured. If a separate data channel is used for each sample, the configuration is *planar*. If data for all samples is stored in a single data channel, the configuration is *interleaved*.

When an alpha sample is present, the number of bits per pixel doubles for planar configurations (4 for NSTwoBitGrayDepth and 16 for NSEightBitGrayDepth). Interleaved configurations already account for an alpha sample whether or not it's present; thus, the number of bits per pixel for NSTwelveBitRGBDepth and NSTwentyFourBitRGBDepth depths remains unchanged.

The constant NSDefaultDepth is also available. If *depth* is NSDefaultDepth, the context's default depth limit is set to the Window Server's maximum visible depth, which is determined by which screens are active.

The **rangecheck** error is executed if *depth* is invalid.

ERRORS **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **currentdefaultdepthlimit**, **setwindowdepthlimit**, **currentwindowdepthlimit**, **currentwindowdepth**

seteventmask

SYNOPSIS *mask window* **seteventmask** –

Warning: Don't use this operator if you're using the Application Kit. Use Window's **setEventMask:** method instead.

Sets the Server-level event mask for the specified window to *mask*. For windows created by the window packages, this mask may allow additional event types beyond those requested by the application. The following operand names are defined for *mask*:

Mask Operand	Event Type Allowed
Lmousedownmask	Mouse-down, left or only mouse button
Lmouseupmask	Mouse-up, left or only mouse button
Rmousedownmask	Mouse-down, right mouse button
Rmouseupmask	Mouse-up, right mouse button
Mousemovedmask	Mouse-moved
Lmousedraggedmask	Mouse-dragged, left or only mouse button
Rmousedraggedmask	Mouse-dragged, right mouse button
Mouseenteredmask	Mouse-entered
Mouseexitedmask	Mouse-exited
Keydownmask	Key-down
Keyupmask	Key-up
Flagschangedmask	Flags-changed
Kitdefinedmask	Kit-defined
Sysdefinedmask	System-defined
Appdefinedmask	Application-defined
Allevents	All event types

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **currenteventmask**

setexposurecolor

SYNOPSIS – **setexposurecolor** –

Applies to nonretained windows only; sets the exposure color to the color specified by the current color parameter in the current graphics state. The exposure color (white by default) determines the color of newly exposed areas of the window and of new areas created by **placewindow**. The alpha value of these areas is always 1 (opaque). If the current device is not a window, this operator executes the **invalidid** error.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **placewindow**, **setautofill**, **setsendexposed**

setflushexposures

SYNOPSIS *flag* **setflushexposures** –

Warning: Don't use this operator if you're using the Application Kit.

Sets whether window-exposed and screen-changed subevents are flushed to clients. If *flag* is *false*, no window-exposed or screen-changed events are flushed to the client until **setflushexposures** is executed with *flag* equal to *true*. By default, window-exposed and screen-changed events are flushed to clients.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

setframebuffertransfer

SYNOPSIS *redproc greenproc blueproc grayproc fbnum* **setframebuffertransfer** –

Warning: This operator should only be used for the development of screen-calibration products.

Sets the framebuffer transfer functions in effect for the framebuffer indexed by fbnum. fbnum ranges from 0 to countframebuffers–1. The framebuffer transfer describes the relationship between the framebuffer values of the display, and the voltage produced to drive the monitor.

The initial four operands define the transfer procedures: Monochrome devices use grayproc (but see the Note below), color devices use the others. The procedures must be allocated in shared virtual memory. In addition, the Window Server assumes that the framebuffer values are directly proportional to screen brightness. This is important for the accuracy of dithering, compositing, and similar calculations.

The default transfer for NeXT Color Displays is

```
{ 1 2.2 div exp } bind dup dup {}
```

Note: setframebuffertransfer is unsupported on the current generation of NeXT monochrome displays.

It's possible to make framebuffer transfer functions persist beyond the lifetime of the Window Server by storing a property in the NetInfo screens database. In the local NetInfo domain, /localconfig/screens holds the configuration information for the screens known to the Window Server (MegaPixel, NeXTdimension, and so on). These specify the layout and activation state of the screen. The NetInfo defaultTransfer property can contain a string of PostScript code suitable for execution by the setframebuffertransfer operator (without the fbnum parameter). For example, the following represents the NetInfo configuration for a NeXTdimension screen with a default gamma of 2.0:

```
localhost:1#  niutil -read  . /localconfig/screens/NeXTdimension
   name: NeXTdimension
   slot: 2
   unit: 0
   defaultTransfer: {1 2.0 div exp } dup dup dup
   bounds: 0 1120 0 832
   active: 1
   _writers: *
```

The **defaultTransfer** property is used to configure the screen each time the Window Server starts up. This allows monitor calibration products to save their settings so the next time the Window Server starts up, the new values will be used. Note that in some cases, the NetInfo configuration state for a monitor will not have **active** equal to 1, although the monitor is being used by the Window Server. If there are no active screens (screens that are explicitly marked as being active), the Window Server uses a suitable default, however, the other NetInfo properties for that screen are ignored. Thus, you must be sure that the screen for which you are adding a **defaultTransfer** value has **active** set to 1.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **setframebuffertransfer**, **countframebuffers**, **framebuffer**

setinstance

SYNOPSIS *flag* **setinstance** –

Sets the instance-drawing mode in the current graphics state on (if *flag* is *true*) or off (if *flag* is *false*).

ERRORS **stackunderflow**, **typecheck**

SEE ALSO **hideinstance**, **newinstance**

setmouse

SYNOPSIS *x y* **setmouse** –

Moves the mouse location (and, correspondingly, the cursor) to (*x*, *y*), given in the current coordinate system. If the current device isn't a window, the **invalidid** error is executed.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **adjustcursor**, **basetocurrent**, **currentmouse**, **screentocurrent**

setowner

SYNOPSIS *context window* **setowner** –

Sets the owning PostScript context of *window* to *context*. The window is terminated automatically when *context* is terminated.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **currentowner**, **termwindow**, **window**

setsendexposed

SYNOPSIS *flag window* **setsendexposed** –

Warning: Don't use this operator if you're using the Application Kit.

Controls whether the Window Server generates a window-exposed subevent (of the kit-defined event) for *window* under the following circumstances:

- Nonretained window: When an area of the window is exposed, or a new area is created by **placewindow**
- Retained or buffered window: When an area of the window that had instance drawing in it is exposed

By default, window-exposed subevents are generated under these circumstances. In any case, the window-exposed subevent isn't flushed to the application until the Window Server receives another event.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **setflushexposures**, **placewindow**, **setautofill**, **setexposurecolor**

setshowpageprocedure

SYNOPSIS *proc window* **setshowpageprocedure** –

Warning: Don't use this operator if you're using the Application Kit.

Sets the PostScript procedure that's executed, for the specified window, when the **showpage** procedure is executed. *proc* must be allocated in shared virtual memory.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **currentshowpageprocedure**

settrackingrect

SYNOPSIS *x y width height leftbool rightbool insidebool userdata trectnum gstate*
settrackingrect –

SYNOPSIS *or*

SYNOPSIS *x y width height optionarray trectnum gstate* **settrackingrect** –

> **Important:** The settrackingrect operator boasts two form, distinguished by the number and contents of the operands that are passed. The operator itself looks at its operands to determine how to proceed. The common portion of the two forms is described immediately below. Attention is then paid to the features that set the forms apart.

Sets a tracking rectangle in the window referred to by *gstate* to the rectangle specified by *x*, *y*, *width*, and *height* (in the coordinate system of that graphics state). If *gstate* is **null**, the window referred to by the current graphics state is used. *trectnum* is an arbitrary integer that can be any number except 0. It's used to identify tracking rectangles; no two tracking rectangles can share the same *trectnum* value. Any number of tracking rectangles may be set in a single window.

The tracking rectangle will remain in effect until **cleartrackingrect** is called, or until another tracking rectangle with the same *trectnum* is set.

Form 1

SYNOPSIS *x y width height leftbool rightbool insidebool userdata trectnum gstate*
settrackingrect –

In this form, the application receives mouse-exited and mouse-entered events as the cursor leaves and reenters the visible portion of the tracking rectangle. In the event record for a mouse-exited or mouse-entered event, the **data** component will contain *trectnum* along with the event number of the last mouse-down event.

userdata is an arbitrary integer that you assign to the tracking rectangle. Since several tracking rectangles can share the same *userdata* value, you can use *userdata* to identify an object in your application that will be notified when a mouse-entered or mouse-exited event occurs in any of the tracking rectangles.

You can specify that mouse-entered and mouse-exited events be generated only if certain mouse buttons are down. If *leftbool* is *true*, the events will be generated only when the left mouse button is down; likewise for *rightbool* and the right mouse button. If both *leftbool* and *rightbool* are *true*, the events will be generated only if both mouse buttons are down. If both *leftbool* and *rightbool* are *false*, the position of the mouse buttons isn't taken into account in generating mouse-entered and mouse-exited events.

settrackingrect causes the Window Server to repeatedly compare the current cursor position to the previous one to see whether the cursor has moved from inside the tracking rectangle to outside it or vice versa. *insidebool* tells **settrackingrect** whether to consider the initial cursor position to be inside or outside the tracking rectangle:

- If *insidebool* is *true* and the cursor is initially outside the tracking rectangle, a mouse-exited event is generated.
- If *insidebool* is *false* and the cursor is initially inside the tracking rectangle, a mouse-entered event is generated.

Form 2

SYNOPSIS *x y width height optionarray trectnum gstate* **settrackingrect** –

In this form, **settrackingrect** sets special event-gathering attributes of a rectangle (events are *not* generated when the boundary is crossed).

optionarray contains key-value pairs that define the attributes that you're interested in. An empty option array is meaningless and will raise a **rangecheck** error. The following keys are currently defined:

Key	Type	Meaning
Pressure	bool	Treat non-zero pressure values as a mouse-down (*false* by default)
Coalesce	bool	Coalesce mouse motion events (*true* by default)

EXAMPLE This example turns pressure on and coalescing off (thereby switching the default values):

```
0 0 10 10 [/Pressure true /Coalesce false] 1 null settrackingrect
```

ERRORS **invalidid**, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **cleartrackingrect**

setwaitcursorenabled

SYNOPSIS *bool context* **setwaitcursorenabled** –

Allows applications to enable and disable wait cursor operation in the specified context. If *context* is 0, **setwaitcursorenabled** sets the global wait cursor flag, which overrides all per-context settings. If the global flag is set to *false*, the wait cursor is disabled for all contexts.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **currentwaitcursorenabled**

setwindowdepthlimit

SYNOPSIS *depth window* **setwindowdepthlimit** –

Warning: Don't use this operator if you're using the Application Kit. Use Window's **setDepthLimit:** method instead.

Sets the depth limit of *window* to *depth*. These depths are defined in **nextdict**:

Depth	Meaning
NSTwoBitGrayDepth	1 spp, 2bps, 2bpp, planar
NSEightBitGrayDepth	1 spp, 8bps, 8bpp, planar
NSTwelveBitRGBDepth	3 spp, 4bps, 16bpp, interleaved
NSTwentyFourBitRGBDepth	3 spp, 8bps, 32bpp, interleaved

where *spp* is the number of samples per pixel; *bps* is the number of bits per sample; and *bpp* is the number of bits per pixel, also know as the window's depth. (The samples-per-pixel value excludes the alpha sample, if present.) *planar* and *interleaved* refer to how the sample data is configured. If a separate data channel is used for each sample, the configuration is *planar*. If data for all samples is stored in a single data channel, the configuration is *interleaved*.

When an alpha sample is present, the number of bits per pixel doubles for planar configurations (4 for NSTwoBitGrayDepth and 16 for NSEightBitGrayDepth). Interleaved configurations already account for an alpha sample whether or not it's present; thus, the number of bits per pixel for NSTwelveBitRGBDepth and NSTwentyFourBitRGBDepth depths remains unchanged.

Another constant, NSDefaultDepth, is defined as the default depth limit in the Window Server's current context. If *depth* is NSDefaultDepth, then the window's depth limit is set to the context's default depth limit. If the resulting depth is lower than the window's current depth, the window's data is dithered down to this depth, which may result in the loss of graphic information.

The **rangecheck** error is executed if *depth* is invalid. The **invalidid** error is executed if *window* doesn't exist.

ERRORS **invalidid**, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **currentwindowdepthlimit**, **setdefaultdepthlimit**, **currentdefaultdepthlimit**, **currentwindowdepth**

setwindowdict

SYNOPSIS *dict window* **setwindowdict** –

Warning: Don't use this operator if you're using the Application Kit.

Sets the dictionary for *window* to *dict*.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **currentwindowdict**

setwindowlevel

SYNOPSIS *level window* **setwindowlevel** –

Sets the window's tier to that specified by *level*. Window tiers constrain the action of the **orderwindow** operator; see **orderwindow** for more information.

You rarely use this operator. To make a panel float above other windows, use the Panel class's **setFloatingPanel:** method.

Attempting to change the level of **workspaceWindow** executes the **invalidaccess** error. (**workspaceWindow** is a PostScript name whose value is the window number of the workspace window.)

ERRORS **invalidaccess**, **invalidid**, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **currentwindowlevel**, **orderwindow**

setwindowtype

SYNOPSIS *type window* **setwindowtype** –

Warning: Don't use this operator if you're using the Application Kit. Use Window's **setBackingType:** method instead.

Sets the window's buffering type to that specified. Currently, the only allowable type conversions are from Buffered to Retained and from Retained to Buffered. All other possibilities execute the **limitcheck** error.

ERRORS **invalidaccess**, **invalidid**, **limitcheck**, **stackunderflow**, **typecheck**

SEE ALSO **window**

setwriteblock

SYNOPSIS *bool* **setwriteblock** –

Sets how the Window Server responds when its output buffer to a client application fills. If *bool* is *true*, the Server defers sending data (event records, error messages, and so on) to that application until there's once again room in the output buffer. In this way, no output data is lost—this is the Server's default behavior. If *bool* is *false*, the Server ignores the state of the output buffer: If the buffer fills and there's more data to be sent, the new data is lost. **setwriteblock** operates on the current context.

Most programmers won't need to use this operator. If you do use it, make sure that you disable the Window Server's default behavior only during the execution of your own PostScript code. If it's disabled while Application Kit code is being executed, errors will result.

ERRORS **stackoverflow**, **typecheck**

SEE ALSO **currentwriteblock**

showcursor

SYNOPSIS – **showcursor** –

Restores the cursor to the screen if it's been hidden with **hidecursor**, unless an outer nested **hidecursor** is still in effect (because it hasn't yet been balanced by a **showcursor**). For example:

```
% cursor is showing initially
. . .
hidecursor       % hides the cursor
. . .
   hidecursor    % cursor stays hidden
   . . .
   showcursor    % cursor still hidden due to first hidecursor
. . .
showcursor       % displays the cursor
```

ERRORS none

SEE ALSO **hidecursor**

showpage

SYNOPSIS – **showpage** –

Warning: This standard operator is different in the OPENSTEP implementation of the Display PostScript system.

This has no effect if the current device is a window; otherwise, it functions as documented by Adobe.

ERRORS none

SEE ALSO **copypage**, **erasepage**

sizeimage

SYNOPSIS *x y width height matrix* **sizeimage** *pixelswide pixelshigh bits/sample matrix multiproc ncolors*

Returns various parameters required by the **readimage** operator when reading the image contained in the rectangle given by *x*, *y*, *width*, and *height* in the current window. (See **readimage** for more information.)

pixelswide and *pixelshigh* are the width and height of the image in pixels. The operand *matrix* is filled with the transformation matrix from user space to the image coordinate system and pushed back on the operand stack.

The other results of this operator describe the window device and are dependent on the window's depth. Each pixel has *ncolors* color components plus one alpha component; the value of each component is described by *bits/sample* bits. If *multiproc* is *true*, **readimage** will need multiple procedures to read the values of the image's pixels. Here are the values that **sizeimage** returns for windows of various depths:

Window Depth	ncolors	bits/sample	multiproc
NSTwoBitGrayDepth	1	2	*true*
NSEightBitGrayDepth	1	8	*true*
NSTwelveBitRGBDepth	3	4	*false*
NSTwentyFourBitRGBDepth	3	8	*false*

ERRORS **stackunderflow**, **typecheck**

SEE ALSO **alphaimage**, **readimage**

stilldown

SYNOPSIS *eventnum* **stilldown** *stilldown*

Returns *true* if the left or only mouse button is still down from the mouse-down event specified by *eventnum*; otherwise it returns *false*. *eventnum* should be the number stored in the **data** component of the event record for an event of type **Lmousedown**.

ERRORS **stackunderflow**, **typecheck**

SEE ALSO **buttondown**, **currentmouse**, **rightbuttondown**, **rightstilldown**

termwindow

SYNOPSIS *window* **termwindow** –

Warning: Don't use this operator if you're using the Application Kit. Use Window's **close** method instead.

Marks *window* for destruction. If the window is in the screen list, it's removed from the screen list and the screen. The given window number will no longer be valid; any attempt to use it will execute the **invalidid** error. The window will actually be destroyed and its storage reclaimed only after the last reference to it from a graphics state is removed. This can be done by resetting the device in the graphics state to another window or to the null device.

Note: After you use the **termwindow** operator, if the terminated window had been the current window, you should use the **nulldevice** operator to remove references to it.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **window**, **windowdevice**, **windowdeviceround**

window

SYNOPSIS *x y width height type* **window** *window*

Warning: Don't use this operator if you're using the Application Kit. Create a Window object instead.

Creates a window that has a lower left corner of (*x*, *y*) and the indicated width and height. *x*, *y*, *width*, and *height* are given in the screen coordinate system. No portion of a window can have an x or y coordinate with an absolute value greater than 16000; *width* and *height* must be in the range from 0 to 10000. Exceeding these limits executes the **rangecheck** error. The four operands can be integer or real numbers; however, they are converted to integers in the Window Server by rounding toward 0. This operator returns the new window's window number, a nonzero integer that's used to refer to the window.

type specifies the window's buffering type as **Buffered**, **Retained**, or **Nonretained**.

The new window won't be in the screen list; you can put it there with the **orderwindow** operator. Windows that aren't in the screen list don't appear on the screen and don't receive user events.

The **window** operator also does the following:

- Sets the origin of the window's base coordinate system to the lower left corner of the window
- Sets the window's clipping path to the outer edge of the window
- Fills the window with opaque white and sets the window's exposure color to white

Note: This operator does not make the new window the current window; to do that, use **windowdeviceround** or **windowdevice**.

ERRORS **invalidid**, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **setexposurecolor**, **termwindow**, **windowdeviceround**

windowdevice

SYNOPSIS *window* **windowdevice** –

Sets the current device of the current graphics state to the given window device. It also sets the origin of the window's default matrix to the lower left corner of the window. One unit in the user coordinate system is made equal to 1/72 of an inch. The clipping path is reset to a rectangle surrounding the window. Other elements of the graphics state remain unchanged. This matrix becomes the default matrix for the window: **initmatrix** will reestablish this matrix.

windowdevice is rarely used in OPENSTEP since the coordinate system it establishes isn't aligned with the pixels on the screen. Use the related operator **windowdeviceround** to create a coordinate system that is aligned.

Don't use this operator lightly, as it creates a new matrix and clipping path. It's significantly more expensive than a **setgstate** operator.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **windowdeviceround**

windowdeviceround

SYNOPSIS *window* **windowdeviceround** –

Sets the current device of the current graphics state to the given window device. It also sets the origin of the window's default matrix to the lower left corner of the window. One unit in the user coordinate system is made equal to the width of one pixel. The clipping path is reset to a rectangle surrounding the window. Other elements of the graphics state remain unchanged. This matrix becomes the default matrix for the window: **initmatrix** will reestablish this matrix.

Don't use this operator blithely, as it creates a new matrix and clipping path. It's significantly more expensive than a **setgstate** operator.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **windowdevice**

windowlist

SYNOPSIS *array context* **windowlist** *subarray*

Fills the array with the window numbers of all windows that are owned by the PostScript context specified by *context*. It returns the subarray containing those window numbers, in order from front to back. If *array* isn't large enough to hold them all, this operator returns the frontmost windows that fit in the array.

EXAMPLE This example yields an array containing the window numbers of all windows that are owned by the current PostScript context:

```
currentcontext
countwindowlist             % find out how many windows
array                       % create array to hold them
currentcontext windowlist   % fill it in
```

ERRORS **stackunderflow**, **typecheck**

SEE ALSO **countscreenlist**, **countwindowlist**, **screenlist**

NSChangeSpelling

Adopted By: NSText

Declared In: AppKit/NSSpellProtocol.h

Protocol Description

This protocol is implemented by objects in the responder chain that can correct a misspelled word. See the description of the NSSpellChecker class for more information.

Instance Methods

changeSpelling:

– (void)**changeSpelling:**(id)*sender*

Implement this method to replace the selected word in the receiver with a corrected version from the Spelling panel. This message is sent by the NSSpellChecker to the object whose text is being checked. To get the corrected spelling, ask the sender for the string value of its selected cell (visible to the user as the text field in the Spelling panel). This method should replace the selected portion of the text with the string that it gets from the NSSpellChecker.

NSColorPickingCustom

Adopted By: NSColorPicker

Declared In: AppKit/NSColorPicking.h

Protocol Description

Together with the NSColorPickingDefault protocol, NSColorPickingCustom provides a way to add color pickers—custom user interfaces for color selection—to an application's NSColorPanel. The NSColorPickingDefault protocol provides basic behavior for a color picker. The NSColorPicker class adopts the NSColorPickingDefault protocol. The easiest way to implement a color picker is to create a subclass of NSColorPicker and implement the NSColorPickingCustom protocol for this new class.

Note: All of NSColorPickingCustom's methods must be implemented by the custom color picker.

Here are the standard color picking modes and mode constants (defined in **AppKit/NSColorPanel.h**):

Mode	Color Mode Constant
Grayscale-Alpha	NSGrayModeColorPanel
Red-Green-Blue	NSRGBModeColorPanel
Cyan-Yellow-Magenta-Black	NSCMYKModeColorPanel
Hue-Saturation-Brightness	NSHSBModeColorPanel
Custom palette	NSCustomPaletteModeColorPanel
Custom color list	NSColorListModeColorPanel
Color wheel	NSWheelModeColorPanel

In grayscale-alpha, red-green-blue, cyan-magenta-yellow-black, and hue-saturation-brightness modes, the user adjusts colors by manipulating sliders. In the custom palette mode, the user can load an NSImage file (TIFF or EPS) into the NSColorPanel, then select colors from the image. In custom color list mode, the user can create and load lists of named colors. The two custom modes provide NSPopUpLists for loading and saving files. Finally, color wheel mode provides a simplified control for selecting colors.

If your color picker includes submodes, you should define a unique value for each submode. As an example, the slider picker has four values defined in the above list (NSGrayModeColorPanel,

NSRGBModeColorPanel, NSCMYKModeColorPanel, and NSHSBModeColorPanel)—one for each of its submodes.

Method Types

Setting the Current Color

– setColor:

Getting the Mode

– currentMode
– supportsMode:

Getting the View

– provideNewView:

Instance Methods

currentMode

– (int)**currentMode**

Returns the color picker's current mode (or submode, if applicable). The returned value should be unique to your color picker. See this protocol description's list of the unique values for the standard color pickers used by the Application Kit.

See also: **– supportsMode:**

provideNewView:

– (NSView *)**provideNewView:**(BOOL)*initialRequest*

Returns the view containing the color picker's user interface. This message is sent to the color picker whenever the color panel attempts to display it. This may be when the panel is first presented, when the user switches pickers, or when the picker is switched through API. The argument *initialRequest* is YES only when this method is first invoked for your color picker. If *initialRequest* is YES, the method should perform any initialization required (such as lazily loading a nib file, initializing the view, or performing any other custom initialization required for your picker). The NSView returned by this method should be set to automatically resize both its width and height.

setColor:

– (void)**setColor:**(NSColor *)*color*

Adjusts the color picker to make *color* the currently selected color. This method is invoked on the current color picker each time NSColorPanel's **setColor:** method is invoked. If *color* is actually different from the color picker's color (as it would be if, for example, the user dragged a color into NSColorPanel's color well), this method could be used to update the color picker's color to reflect the change.

supportsMode:

– (BOOL)**supportsMode:**(int)*mode*

Returns whether or not the receiver supports the specified picking mode. This method is invoked when the NSColorPanel's is first initialized: It is used to attempt to restore the user's previously selected mode. It is also invoked by NSColorPanel's **setMode:** method to find the color picker that supports a particular mode. See this protocol description's list of the unique mode values for the standard color pickers used by the Application Kit.

See also: **– currentMode**

NSColorPickingDefault

Adopted By: NSColorPicker

Declared In: AppKit/NSColorPicking.h

Protocol Description

The NSColorPickingDefault protocol, together with the NSColorPickingCustom protocol, provides an interface for adding color pickers—custom user interfaces for color selection—to an application's NSColorPanel. The NSColorPickingDefault protocol provides basic behavior for a color picker. The NSColorPickingCustom protocol provides implementation-specific behavior.

The NSColorPicker class implements the NSColorPickingDefault protocol. The simplest way to implement your own color picker is to create a subclass of NSColorPicker, implementing the NSColorPickingCustom protocol for that subclass. However, it's possible to create a subclass of another class, such as NSView, and use it as a base upon which to add the methods of both NSColorPickingDefault and NSColorPickingCustom.

Color Picker Bundles

A class that implements the NSColorPickingDefault and NSColorPickingCustom protocols needs to be compiled and linked in an application's object file. However, your application need not explicitly create an instance of this class. Instead, your application's file package should include a directory named **ColorPickers**; within this directory you should place a directory *MyPickerClass***.bundle** for each custom color picker your application implements. This bundle should contain all resources required for your color picker: nib files, TIFF files, and so on.

NSColorPanel will allocate and initialize an instance of each class for which a bundle is found in the **ColorPickers** directory. The class name is assumed to be the bundle directory name minus the **.bundle** extension.

Color Picker Buttons

NSColorPanel lets the user select a color picker from an NSMatrix of NSButtonCells. This protocol includes methods for providing and manipulating the image that gets displayed on the button.

Color Mask and Color Modes

The color mask determines which color mode is enabled for NSColorPanel. This mask is set before you initialize a new instance of NSColorPanel. NSColorPanelAllModesMask represents the logical OR of the

other color mask constants: It causes the NSColorPanel to display all standard color pickers. When initializing a new instance of NSColorPanel, you can logically OR any combination of color mask constants to restrict the available color modes. The predefined color mask constants are:

Mode	Color Mask Constant
Grayscale-Alpha	NSColorPanelGrayModeMask
Red-Green-Blue	NSColorPanelRGBModeMask
Cyan-Yellow-Magenta-Black	NSColorPanelCMYKModeMask
Hue-Saturation-Brightness	NSColorPanelHSBModeMask
Custom palette	NSColorPanelCustomPaletteModeMask
Custom color list	NSColorPanelColorListModeMask
Color wheel	NSColorPanelWheelModeMask
All of the above	NSColorPanelAllModesMask

When an application's instance of NSColorPanel is masked for more than one color mode, your program can set its active mode by invoking the **setMode:** method with a color mode constant as its argument; the user can set the mode by clicking buttons on the panel. Here are the standard color modes and mode constants:

Mode	Color Mode Constant
Grayscale-Alpha	NSGrayModeColorPanel
Red-Green-Blue	NSRGBModeColorPanel
Cyan-Yellow-Magenta-Black	NSCMYKModeColorPanel
Hue-Saturation-Brightness	NSHSBModeColorPanel
Custom palette	NSCustomPaletteModeColorPanel
Custom color list	NSColorListModeColorPanel
Color wheel	NSWheelModeColorPanel

In grayscale-alpha, red-green-blue, cyan-magenta-yellow-black, and hue-saturation-brightness modes, the user adjusts colors by manipulating sliders. In the custom palette mode, the user can load an NSImage file (TIFF or EPS) into the NSColorPanel, then select colors from the image. In custom color list mode, the user can create and load lists of named colors. The two custom modes provide NSPopUpLists for loading and saving files. Finally, color wheel mode provides a simplified control for selecting colors.

These constants are defined in **AppKit/NSColorPanel.h**.

Method Types

Initializing a Color Picker
– initWithPickerMask:colorPanel:

Setting the Mode
– setMode:

Using Color Lists
– attachColorList:
– detachColorList:

Adding Button Images
– insertNewButtonImage:in:
– provideNewButtonImage

Showing Opacity Controls
– alphaControlAddedOrRemoved:

Responding to a Resized View
– viewSizeChanged:

Instance Methods

alphaControlAddedOrRemoved:

– (void)**alphaControlAddedOrRemoved:**(id)*sender*

Sent by the color panel when the opacity controls have been hidden or displayed. Invoked automatically when the NSColorPanel's opacity slider is added or removed; you never invoke this method directly.

If the color picker has its own opacity controls, it should hide or display them, depending on whether the sender's **showsAlpha** method returns NO or YES.

attachColorList:

– (void)**attachColorList:**(NSColorList *)*colorList*

Tells the color picker to attach the given *colorList*, if it isn't already displaying the list. You never invoke this method; it's invoked automatically by the NSColorPanel when its **attachColorList:** method is invoked. Since NSColorPanel's list mode manages NSColorLists, this method need only be implemented by a custom color picker that manages NSColorLists itself. This method ordinarily doesn't do anything, since NSColorPanel's list mode manages NSColorLists.

See also: **– detachColorList:**

detachColorList:

– (void)**detachColorList:**(NSColorList *)*colorList*

Tells the color picker to detach the given *colorList*, unless the receiver isn't displaying the list. You never invoke this method; it's invoked automatically by the NSColorPanel when its **detachColorList:** method is invoked. Since NSColorPanel's list mode manages NSColorLists, this method need only be implemented by a custom color picker that manages NSColorLists itself. This method ordinarily doesn't do anything, since NSColorPanel's list mode manages NSColorLists.

See also: **– attachColorList:**

initWithPickerMask:colorPanel:

– (id)**initWithPickerMask:**(int)*mask*
colorPanel:(NSColorPanel *)*owningColorPanel*

Notifies the color picker of the color panel's mask and initializes the color picker. This method is sent by the NSColorPanel to all implementors of the color picking protocols when the application's color panel is first initialized. In order for your color picker to receive this message, it must have a bundle in your application's "ColorPickers" directory (described in "Color Picker Bundles" in the Protocol Description).

mask is determined by the argument to the NSColorPanel method **setPickerMask:**. If no mask has been set, *mask* is NSColorPanelAllModesMask. If your color picker supports any additional modes, you should invoke the **setPickerMask:** method when your application initializes to notify the NSColorPanel class. The standard mask constants are:

Mode	Color Mask Constant
Grayscale-Alpha	NSColorPanelGrayModeMask
Red-Green-Blue	NSColorPanelRGBModeMask

Mode	Color Mask Constant
Cyan-Yellow-Magenta-Black	NSColorPanelCMYKModeMask
Hue-Saturation-Brightness	NSColorPanelHSBModeMask
Custom palette	NSColorPanelCustomPaletteModeMask
Custom color list	NSColorPanelColorListModeMask
Color wheel	NSColorPanelWheelModeMask
All of the above	NSColorPanelAllModesMask

This method should examine the mask and determine whether it supports any of the modes included there. You may also check the value in *mask* to enable or disable any subpickers or optional controls implemented by your color picker. Your color picker may also retain *owningColorPanel* in an instance variable for future communication with the color panel.

This method is provided to initialize your color picker; however, much of a color picker's initialization may be done lazily through the NSColorPickingCustom protocol's **provideNewView:** method. If your color picker responds to any of the modes represented in *mask*, it should perform its initialization and return **self**. Color pickers that do so have their buttons inserted in the color panel and continue to receive messages from the panel as the user manipulates it. If the color picker doesn't respond to any of the modes represented in mask, it should do nothing and return **nil**.

See also: **+ setPickerMask:** (NSColorPanel class)

insertNewButtonImage:in:

– (void)**insertNewButtonImage:**(NSImage *)*newButtonImage*
in:(NSButtonCell *)*buttonCell*

Sets *newButtonImage* as *buttonCell*'s image. *buttonCell* is the NSButtonCell object that lets the user choose the picker from the color panel—the color picker's representation in the NSColorPanel's picker NSMatrix. This method should perform application-specific manipulation of the image before it's inserted and displayed by the button cell.

See also: **– provideNewButtonImage**

provideNewButtonImage

– (NSImage *)**provideNewButtonImage**

Returns the image for the mode button that the user uses to select this picker in the color panel, that is, the color picker's representation in the NSColorPanel's picker NSMatrix. (This is the same image that the color panel uses as an argument when sending the **insertNewButtonImage:in:** message.)

setMode:

– (void)**setMode:**(int)*mode*

Sets the color picker's mode. This method is invoked by NSColorPanel's **setMode:** method to ensure that the color picker reflects the current mode. For example, invoke this method during color picker initialization to ensure that all color pickers are restored to the mode the user left them in the last time an NSColorPanel was used.

Most color pickers have only one mode, and thus don't need to do any work in this method. An example of a color picker that uses this method is the slider picker, which can choose from one of several submodes depending on the value of *mode*. The available modes are:

Mode	Color Mode Constant
Grayscale-Alpha	NSGrayModeColorPanel
Red-Green-Blue	NSRGBModeColorPanel
Cyan-Yellow-Magenta-Black	NSCMYKModeColorPanel
Hue-Saturation-Brightness	NSHSBModeColorPanel
Custom palette	NSCustomPaletteModeColorPanel
Custom color list	NSColorListModeColorPanel
Color wheel	NSWheelModeColorPanel

viewSizeChanged:

– (void)**viewSizeChanged:**(id)*sender*

Tells the color picker when the NSColorPanel's view size changes in a way that might affect the color picker. *sender* is the NSColorPanel that contains the color picker. Use this method to perform special preparation when resizing the color picker's view. Since this method is invoked only as appropriate, it's

better to implement this method than to override the method **superviewSizeChanged:** for the NSView in which the color picker's user interface is contained.

See also: **– provideNewView:** (NSColorPickingCustom protocol)

NSComboBoxCellDataSource

(Informal Protocol)

Category Of: NSObject

Declared In: AppKit/NSComboBoxCell.h

Category Description

The NSComboBoxCellDataSource category declares the methods that an NSComboBoxCell uses to access the contents of its data source object. The combo box cell determines how many items to display by sending a **numberOfItemsInComboBoxCell:** message, and accesses individual values with the **comboBoxCell: objectValueForItemAtIndex:** method. Incremental searches—performed when a user types into the combo box's text field while the pop-up list is displayed—are performed by sending **comboBoxCell: indexOfItemWithStringValue:** messages to the combo box cell's data source.

The NSComboBoxCell treats objects provided by its data source as values to be displayed in the combo box's pop-up list. If these objects aren't of common value classes—such as NSString, NSNumber, and so on—you'll need to create a custom NSFormatter to display them. See the NSFormatter class specification for more information.

When an NSComboBoxCellDataSource is asked to supply a data item, the NSComboBoxCell that sends the request is provided as a parameter. This allows a single data source object to manage several sets of data, choosing the appropriate set based on the identify of the NSComboBoxCell that sends the message.

Instance Methods

comboBoxCell:indexOfItemWithStringValue:

– (unsigned int)**comboBoxCell:**(NSComboBoxCell *)*aComboBoxCell*
indexOfItemWithStringValue:(NSString *)*aString*

An NSComboBoxCell uses this method to perform incremental—or "smart"—searched when the user types into the text field with the pop-up list displayed. Your implementation of this method should return the index for the item which matches *aString*, or NSNotFound if no item matches. This method is optional; if you don't provide an implementation for this method, no searches occur.

comboBoxCell:objectValueForItemAtIndex:

– (id)**comboBoxCell:**(NSComboBoxCell *)*aComboBoxCell* **objectValueForItemAtIndex:**(int)*index*

Implement this method to return the object that corresponds to the item at *index* in *aComboBoxCell*. Your data source must implement this method.

numberOfItemsInComboBoxCell:

– (int)**numberOfItemsInComboBoxCell:**(NSComboBoxCell *)*aComboBoxCell*

Implement this method to return the number of items managed for *aComboBoxCell* by your data source object. An NSComboBoxCell uses this method to determine how many items it should display in its pop-up list. Your data source must implement this method.

NSComboBoxDataSource

(Informal Protocol)

Category Of: NSObject

Declared In: AppKit/NSComboBox.h

Category Description

The NSComboBoxDataSource category declares the methods that an NSComboBox uses to access the contents of its data source object. The combo box determines how many items to display by sending a **numberOfItemsInComboBox:** message, and accesses individual values with the **comboBox: objectValueForItemAtIndex:** method. Incremental searches—performed when a user types into the combo box's text field while the pop-up list is displayed—are performed by sending **comboBox: indexOfItemWithStringValue:** messages to the combo box's data source.

The NSComboBox treats objects provided by its data source as values to be displayed in the combo box's pop-up list. If these objects aren't of common value classes—such as NSString, NSNumber, and so on—you'll need to create a custom NSFormatter to display them. See the NSFormatter class specification for more information.

When an NSComboBoxDataSource is asked to supply a data item, the NSComboBox that sends the request is provided as a parameter. This allows a single data source object to manage several sets of data, choosing the appropriate set based on the identify of the NSComboBox that sends the message.

Instance Methods

comboBox:indexOfItemWithStringValue:

– (unsigned int)**comboBox:**(NSComboBox *)*aComboBox* **indexOfItemWithStringValue:** (NSString *)*aString*

An NSComboBox uses this method to perform incremental—or "smart"—searched when the user types into the text field with the pop-up list displayed. Your implementation of this method should return the index for the item which matches *aString*, or NSNotFound if no item matches. This method is optional; if you don't provide an implementation for this method, no searches occur.

comboBox:objectValueForItemAtIndex:

– (id)**comboBox:**(NSComboBox *)*aComboBox* **objectValueForItemAtIndex:**(int)*index*

Implement this method to return the object that corresponds to the item at *index* in *aComboBox*. Your data source must implement this method.

numberOfItemsInComboBox:

– (int)**numberOfItemsInComboBox:**(NSComboBox *)*aComboBox*

Implement this method to return the number of items managed for *aComboBox* by your data source object. An NSComboBox uses this method to determine how many items it should display in its pop-up list. Your data source must implement this method.

NSDPSContextNotification

Adopted By: no OpenStep classes

Declared In: AppKit/NSDPSContext.h

Protocol Description

The NSDPSContextNotification protocol supplies information about the execution status of a sequence of PostScript commands previously sent to the Display PostScript server.

Instance Methods

contextFinishedExecuting:

– (void)**contextFinishedExecuting:**(NSDPSContext *)*context*

Notifies the receiver that the context has finished executing a batch of PostScript commands. See **notifyObjectWhenFinishedExecuting:** (NSDPSContext).

NSDraggingDestination

(informal protocol)

Category Of: NSObject

Declared In: AppKit/NSDragging.h

Description

The NSDraggingDestination informal protocol declares methods that the destination (or recipient) of a dragged image must implement. The destination automatically receives NSDraggingDestination messages as an image enters, moves around inside, and then exits or is released within the destination's boundaries.

In the text here and in the other dragging protocol descriptions, the term *dragging session* is the entire process during which an image is selected, dragged, released, and absorbed or rejected by the destination. A *dragging operation* is the action that the destination takes in absorbing the image when it's released. The *dragging source* is the object that "owns" the image that's being dragged. It's specified as an argument to the **dragImage:at:offset:event:pasteboard:source:slideBack:** message, sent to a window or view object, that instigated the dragging session.

The Dragged Image

The image that's dragged in an image-dragging session is simply an image that represents data that resides on the pasteboard. Although a dragging destination can access the image (through the **draggedImage** method described in the NSDraggingInfo protocol), its primary concern is with the pasteboard data that the image represents—the dragging operation that a destination ultimately performs is on the pasteboard data, not on the image itself.

Valid Destinations

Dragging is a visual phenomenon. To be an image-dragging destination, an object must represent a portion of screen real estate; thus, only window and view objects can be destinations. Furthermore, you must register the pasteboard types that the object will accept by sending the object a **registerForDraggedTypes:** message, defined in both NSWindow and NSView. During a dragging session, a candidate destination only receives NSDraggingDestination messages if the destination is registered for a pasteboard type that matches the type of the pasteboard data being dragged. See the NSPasteboard class specification for more information about pasteboard types.

Although NSDraggingDestination is declared as an informal protocol, the NSWindow and NSView subclasses that you create to adopt the protocol need only implement those methods that are pertinent. (The NSWindow and NSView classes provide private implementations for all of the methods.) Either a window

object or its delegate may implement these methods; however, the delegate's implementation takes precedence if there are implementations in both places.

The Sender of Destination Messages

Each of the NSDraggingDestination methods sports a single argument: *sender*, the object that invoked the method. Within its implementations of the NSDraggingDestination methods, the destination can send NSDraggingInfo protocol messages to *sender* to get more information on the current dragging session.

The Order of Destination Messages

The six NSDraggingDestination methods are invoked in a distinct order:

- As the image is dragged into the destination's boundaries, the destination is sent a **draggingEntered:** message
- While the image remains within the destination, a series of **draggingUpdated:** messages are sent.
- If the image is dragged out of the destination, **draggingExited:** is sent and the sequence of NSDraggingDestination messages stops. If it re-enters, the sequence begins again (with a new **draggingEntered:** message).
- When the image is released, it either slides back to its source (and breaks the sequence) or a **prepareForDragOperation:** message is sent to the destination, depending on the value returned by the most recent invocation of **draggingEntered:** or **draggingUpdated:**.
- If the **prepareForDragOperation:** message returned YES, a **performDragOperation:** message is sent.
- Finally, if **performDragOperation:** returned YES, **concludeDragOperation:** is sent.

Method Types

Before the image is released

– draggingEntered:
– draggingUpdated:
– draggingExited:

After the image is released

– prepareForDragOperation:
– performDragOperation:
– concludeDragOperation:

Instance Methods

concludeDragOperation:

– (void)**concludeDragOperation:**(id <NSDraggingInfo>)*sender*

Invoked when the dragging operation is complete and the previous **performDragOperation:** returned YES. The destination implements this method to perform any tidying up that it needs to do, such as updating its visual representation now that it has incorporated the dragged data. This is the last message that's sent from *sender* to the destination during a dragging session.

draggingEntered:

– (unsigned int)**draggingEntered:**(id <NSDraggingInfo>)*sender*

Invoked when a dragged image enters the destination. Specifically, this method is invoked when the mouse pointer enters the destination's bounds rectangle (if it's a view object) or its frame rectangle (if it's a window object).

This method must return a value that indicates which dragging operation the destination will perform when the image is released. In deciding which dragging operation to return, the method should evaluate the overlap between both the dragging operations allowed by the source (accessible through the **draggingSourceOperationMask** method) and the dragging operations and pasteboard data types that the destination itself supports. The returned value should be exactly one of the following:

Option	Meaning
NSDragOperationCopy	The data represented by the image will be copied.
NSDragOperationLink	The data will be shared.
NSDragOperationGeneric	The operation will be defined by the destination.
NSDragOperationPrivate	The operation is negotiated privately between the source and the destination.
NSDragOperationAll	Combines all the above.

If none of the operations is appropriate, this method should return NSDragOperationNone (this is the default response if the method isn't implemented by the destination).

The code below is a simple example of a method that responds distinctly when one of two different types of data is dragged into the destination view or window. If the dragged data is a color and the source object permits copying, the return value indicates that the destination will permit copying of the color data on the pasteboard. If the dragged data is an RTF file and the source object permits linking, the return value

indicates that the destination will permit linking of the RTF file on the pasteboard. Otherwise the code returns NSDragOperationNone, indicating that the destination will not permit any dragging operations with the data on pasteboard.

```
- (unsigned int)draggingEntered:(id <NSDraggingInfo>)sender
{
    NSPasteboard *pboard;
    NSDragOperation sourceDragMask;

    sourceDragMask = [sender draggingSourceOperationMask];
    pboard = [sender draggingPasteboard];

    if ([[pboard types] indexOfObject:NSColorPboardType] != NSNotFound) {
        if (sourceDragMask & NSDragOperationCopy) {
            return NSDragOperationCopy;
        }
    }
    if ([[pboard types] indexOfObject:NSRTFPboardType] != NSNotFound) {
        if (sourceDragMask & NSDragOperationLink) {
            return NSDragOperationLink;
        }
    }
    return NSDragOperationNone;
}
```

See also: – **draggingUpdated:**, – **draggingExited:**, – **prepareForDragOperation:**

draggingExited:

– (void)**draggingExited:**(id <NSDraggingInfo>)*sender*

Invoked when the dragged image exits the destination's bounds rectangle (in the case of a view object) or its frame rectangle (in the case of a window object).

draggingUpdated:

– (unsigned int)**draggingUpdated:**(id <NSDraggingInfo>)*sender*

Invoked periodically as the image is held within the destination. The messages continue until the image is either released or dragged out of the window or view. The return value should be one of the dragging operation options listed under the **draggingEntered:** method. The default return value (if this method isn't implemented by the destination) is the value returned by the previous **draggingEntered:** message.

This method provides the destination with an opportunity to modify the dragging operation depending on the position of the mouse pointer inside of the destination view or window object. For example, you may have several graphics or areas of text contained within the same view and wish to tailor the dragging

operation, or to ignore the drag event completely, depending upon which object is underneath the mouse pointer at the time when the user releases the dragged image and the **performDragOperation:** method is invoked.

You typically examine the contents of the pasteboard in the **draggingEntered:** method, where this examination is performed only once, rather than in the **draggingUpdated:** method, which is invoked multiple times.

Only one destination at a time receives a sequence of **draggingUpdated:** messages. If the mouse pointer is within the bounds of two overlapping views that are both valid destinations, the uppermost view receives these messages until the image is either released or dragged out.

See also: – **draggingExited:**, – **prepareForDragOperation:**

performDragOperation:

– (BOOL)**performDragOperation:**(id <NSDraggingInfo>)*sender*

Invoked after the released image has been removed from the screen and the previous **prepareForDragOperation:** message has returned YES. The destination should implement this method to do the real work of importing the pasteboard data represented by the image. If the destination accepts the data, it returns YES, otherwise it returns NO. The default is to return NO.

See also: – **concludeDragOperation:**

prepareForDragOperation:

– (BOOL)**prepareForDragOperation:**(id <NSDraggingInfo>)*sender*

Invoked when the image is released, if the most recent **draggingEntered:** or **draggingUpdated:** message returned an acceptable drag-operation value. Returns YES if the receiver agrees to perform the drag operation and NO if not.

See also: – **performDragOperation:**

NSDraggingInfo

Adopted By: no Rhapsody classes

Declared In: AppKit/NSDragging.h

Protocol Description

The NSDraggingInfo protocol declares methods that supply information about a *dragging session* (see the NSDraggingDestination protocol for definitions of dragging terms). NSDraggingInfo methods are designed to be invoked from within a class's implementation of NSDraggingDestination informal protocol methods. The Application Kit automatically passes an object that conforms to the NSDraggingInfo protocol as the argument to each of the methods defined by NSDraggingDestination. NSDraggingInfo messages should be sent to this object; you never need to create a class that implements the NSDraggingInfo protocol.

Method Types

Dragging-session information

– draggingSource
– draggingSourceOperationMask
– draggingDestinationWindow
– draggingPasteboard
– draggingSequenceNumber
– draggingLocation

Image information

– draggedImage
– draggedImageLocation

Sliding the image

– slideDraggedImageTo:

Instance Methods

draggedImage

– (NSImage *)**draggedImage**

Returns the image being dragged. This image object visually represents the data put on the pasteboard during the drag operation; however, it is the pasteboard data and not this image that are ultimately utilized in the dragging operation.

See also: – **draggedImageLocation**

draggedImageLocation

– (NSPoint)**draggedImageLocation**

Returns the current location of the dragged image's origin in the base coordinate system of the destination object's window. The image moves along with the mouse pointer (the position of which is given by **draggingLocation**) but may be positioned at some offset.

See also: – **draggedImage**

draggingDestinationWindow

– (NSWindow *)**draggingDestinationWindow**

Returns the destination window for the dragging operation. Either this window is the destination itself, or it contains the view object that is the destination.

draggingLocation

– (NSPoint)**draggingLocation**

Returns the current location of the mouse pointer in the base coordinate system of the destination object's window.

See also: – **draggedImageLocation**

draggingPasteboard

– (NSPasteboard *)**draggingPasteboard**

Returns the pasteboard object that holds the data being dragged. The dragging operation that is ultimately performed utilizes this pasteboard data and not the image returned by the **draggedImage** method.

draggingSequenceNumber

– (int)**draggingSequenceNumber**

Returns a number that uniquely identifies the dragging session.

draggingSource

– (id)**draggingSource**

Returns the source, or owner, of the dragged data or **nil** if the source isn't in the same application as the destination. The dragging source implements methods from the NSDraggingSource informal protocol.

draggingSourceOperationMask

– (unsigned int)**draggingSourceOperationMask**

Returns the dragging operation mask declared by the dragging source (through its **draggingSourceOperationMaskForLocal:** method). If the source permits dragging operations, the elements in the mask will be one or more of the following, combined using the C bitwise OR operator:

Option	Meaning
NSDragOperationCopy	The data represented by the image can be copied.
NSDragOperationLink	The data can be shared.
NSDragOperationGeneric	The operation can be defined by the destination.
NSDragOperationPrivate	The operation is negotiated privately between the source and the destination.
NSDragOperationAll	Combines all the above.

If the source does not permit any dragging operations, then method should return NSDragOperationNone.

If the user is holding down a modifier key during the dragging session and the source doesn't prohibit modifier keys from affecting the drag operation (through its **ignoreModifierKeysWhileDragging** method), then the operating system combines the dragging operation value that corresponds to the modifier key (see the descriptions below) with the source's mask using the C bitwise AND operator.

On Mach the modifier keys are associated with the dragging operation options shown below,

Modifier Key	Dragging Option
Control	NSDragOperationLink
Alternate	NSDragOperationCopy
Command	NSDragOperationGeneric

while on Windows the modifier keys are associated with the following dragging operation options.

Modifier Key	Dragging Option
Control	NSDragOperationCopy
Shift-Control	NSDragOperationLink
Alternate	NSDragOperationCopy

slideDraggedImageTo:

– (void)**slideDraggedImageTo:**(NSPoint)*aPoint*

Slides the image to *aPoint*, a specified location in the screen coordinate system. This method can be used to snap the image down to a particular location. It should only be invoked from within the destination's implementation of **prepareForDragOperation:**—in other words, after the user has released the image but before it's removed from the screen.

NSDraggingSource

(informal protocol)

Category Of: NSObject

Declared In: AppKit/NSDragging.h

Description

The NSDraggingSource informal protocol declares methods that are implemented by the source object in a dragging session (see the NSDraggingDestination protocol for definitions of dragging terms). The *dragging source* is specified as an argument to the **dragImage:at:offset:event:pasteboard:source: slideBack:** message, sent to a window or view object to initiate the dragging session.

Of the methods declared below, only **draggingSourceOperationMaskForLocal:** must be implemented. The other methods are invoked only if the dragging source implements them. All four methods are invoked automatically during a dragging session—you never send an NSDraggingSource message directly to an object.

Method Types

Specifying dragging options

– draggingSourceOperationMaskForLocal:
– ignoreModifierKeysWhileDragging

Responding to dragging sessions

– draggedImage:beganAt:
– draggedImage:endedAt:deposited:

Instance Methods

draggedImage:beganAt:

– (void)**draggedImage:**(NSImage *)*anImage* **beganAt:**(NSPoint)*aPoint*

Invoked when *anImage* is displayed but before it starts following the mouse. *aPoint* is the origin of the image in screen coordinates. This method provides the source object with an opportunity to respond to the initiation of a dragging session. For example, you might choose to have the source give a visual indication to the user that data is being dragged from the source.

See also: **– convertScreenToBase:** (NSWindow), **– convertBaseToScreen:** (NSWindow), **– convertPoint:fromView:** (NSView), **– convertPoint:toView:** (NSView)

draggedImage:endedAt:deposited:

– (void)**draggedImage:**(NSImage *)*anImage*
endedAt:(NSPoint)*aPoint*
deposited:(BOOL)*flag*

Invoked after *anImage* has been released and the dragging destination has been given a chance to operate on the data it represents. *aPoint* is the location of the image's origin in the screen coordinate system when it was released. A YES value for *flag* indicates that the destination accepted the dragged data, while a NO value indicates that it was rejected.

This method provides the source object with an opportunity to respond to either a successful or a failed dragging session. For example, if you are moving data from one location to another, you could use this method to make the source data disappear from its previous location, if the dragging session is successful, or reset itself to its previous state, in the event of a failure.

See also: **– convertScreenToBase:** (NSWindow), **– convertBaseToScreen:** (NSWindow), **– convertPoint:fromView:** (NSView), **– convertPoint:toView:** (NSView)

draggingSourceOperationMaskForLocal:

– (unsigned int)**draggingSourceOperationMaskForLocal:**(BOOL)*flag*

This is the only NSDraggingSource method that must be implemented by the source object. It should return a mask, built by combining the applicable constants listed below using the C bitwise OR operator. You should use this mask to indicate which types of dragging operations the source object will allow to be performed on the dragged image's data. A YES value for *flag* indicates that the candidate destination object (the window or view over which the dragged image is currently poised) is in the same application as the source, while a NO value indicates that the destination object is in a different application.

Option	Meaning
NSDragOperationCopy	The data represented by the image can be copied.
NSDragOperationLink	The data can be shared.
NSDragOperationGeneric	The operation can be defined by the destination.
NSDragOperationPrivate	The operation is negotiated privately between the source and the destination.
NSDragOperationAll	Combines all the above.

If the source does not permit any dragging operations, then it should return NSDragOperationNone.

ignoreModifierKeysWhileDragging

– (BOOL)**ignoreModifierKeysWhileDragging**

Sets whether the use of the modifier keys should have no effect on the type of operation performed. If this method is not implemented or returns NO, then the user can tailor the drag operation by holding down a modifier key during the drag. The dragging option that corresponds to the modifier key is combined with the source's mask (as set with the **draggingSourceOperationMaskForLocal:** method) using the C bitwise AND operator. See the description for the **draggingSourceOperationMask** method in the NSDraggingInfo protocol specification for more information about dragging masks and modifier keys.

NSIgnoreMisspelledWords

Adopted By: NSText

Declared In: AppKit/NSSpellProtocol.h

Protocol Description

Implement this protocol to have the Ignore button in the Spelling panel function properly. The Ignore button allows the user to accept a word that the spelling checker believes is misspelled. In order for this action to update the “ignored words” list for the document being checked, the NSIgnoreMisspelledWords protocol must be implemented.

This protocol is necessary because a list of ignored words is useful only if it pertains to the entire document being checked, but the spelling checker (NSSpellChecker object) does not check the entire document for spelling at once. The spelling checker returns as soon as it finds a misspelled word. Thus, it checks only a subset of the document at any one time. The user usually wants to check the entire document, so usually several spelling checks are run in succession until no misspelled words are found. This protocol allows the list of ignored words to be maintained per-document, even though the spelling checks are not run per-document.

The NSIgnoreMisspelledWords protocol specifies a method, **ignoreSpelling:**, which should be implemented like this:

```
- (void)ignoreSpelling:(id)sender
{
    [[NSSpellChecker sharedSpellChecker] ignoreWord:[[sender selectedCell]
        stringValue] inSpellDocumentWithTag:myDocumentTag];
}
```

The second argument to the NSSpellChecker method **ignoreWord:inSpellDocumentWithTag:** is a tag that the NSSpellChecker can use to distinguish the documents being checked. (See the discussion of “Matching a List of Ignored Words With the Document It Belongs To” in the description of the NSSpellChecker class.) Once the NSSpellChecker has a way to distinguish the various documents, it can append new ignored words to the appropriate list.

To make the ignored words feature useful, the application must store a document’s ignored words list with the document. See the NSSpellChecker class description for more information.

Instance Methods

ignoreSpelling:

– (void)**ignoreSpelling:**(id)*sender*

Implement to allow an application to ignore misspelled words on a document-by-document basis. This message is sent by the NSSpellChecker instance to the object whose text is being checked.

Implement this method by using the code shown in the protocol description.

NSMenuItem

Adopted By: NSMenuItem

Conforms To: NSCoding,
NSCopying
NSObject (NSObject)

Declared In: AppKit/NSMenuItem.h

Warning: The NSMenuItem protocol will be removed from the Application Kit in the Premier release of Rhapsody. The NSMenuItem class will solely assume all associated functionality. This change does not affect binary compatibility between different versions of projects, but might cause failures in project builds. To adapt your projects to this change, alter all references to the protocol (for example, “id <NSMenuItem>”) to references to the class (“NSMenuItem”).

Protocol Description

The NSMenuItem protocol declares methods that are used to manipulate command items in menus. The NSMenuItem class adopts this protocol, implementing all methods the protocol declares, and provides the basic functionality of command items. With some implementations of the OpenStep specification (including OPENSTEP), you cannot replace the NSMenuItem class with a different class which conforms to the NSMenuItem protocol. You may, however, subclass the NSMenuItem class if necessary.

The methods declared by the NSMenuItem protocol allow you to set the titles, actions, targets, tags, images, enabled states, and similar attributes of individual menu items, as well as to obtain the current values of these attributes. As implemented for the NSMenuItem class, a menu item, whenever one of its attributes changes, notifies the associated NSMenu via the **itemChanged:** method. The protocol also allows a conforming object to set keyboard equivalents and (for Microsoft Windows) mnemonics for menu items. See the sections below for more on this functionality.

See the NSMenu, NSMenuView, and NSMenuItemCell class specifications and the NSMenuValidation protocol specification for more information on menus.

Keyboard Equivalents

An object conforming to the NSMenuItem protocol can be assigned a keyboard eqivalent, so that when the user types a character the menu item’s action is sent. The keyboard eqivalent is defined in two parts. First is the basic key equivalent, which must be a Unicode character that can be generated by a single key press without modifier keys (Shift excepted). It is also possible to use a sequence of Unicode characters so long

as the user's key mapping is able to generate the sequence with a single key press. The basic key equivalent is set using **setKeyEquivalent:** and returned by **keyEquivalent**. The second part defines the modifier keys that must also be pressed. This is set using **setKeyEquivalentModifierMask:** and returned by **keyEquivalentModifierMask**. The modifier mask by default includes NSCommandKeyMask, and may also include the masks for the Shift, Alternate, or other modifier keys. Specifying keyboard equivalents in two parts allows you to define a modified keyboard equivalent without having to know which character is generated by the basic key plus the modifier. For example, you can define the keyboard equivalent Command-Alt-f without having to know which character is generated by typing Alt-f.

Certain methods in the NSMenuItem protocol can override assigned keyboard equivalents with those the user has specified in the defaults system. The **setUsesUserKeyEquivalents:** protocol method turns this behavior on or off, and **usesUserKeyEquivalents** returns its status. To determine the user-defined key equivalent for an NSMenuItem object, invoke the **userKeyEquivalent** instance method. If user-defined key equivalents are active and an NSMenuItem object has a user-defined key equivalent, its **keyEquivalent** method returns the user-defined key equivalent and not the one set using **setKeyEquivalent:**.

Mnemonics

On certain platforms, currently including Microsoft Windows, an object conforming to the NSMenuItem protocol can also be assigned a mnemonic. Mnemonics can be assigned on other platforms as well, however, they won't have any effect. Mnemonics are represented by an underlined character in the title of a menu item. The mnemonic can be any character that can be generated by a single key press without modifier keys (Shift excepted). When the menu is active, the user can type the underlined character in the menu item in order to activate that menu item. On Microsoft Windows a user activates the menu by pressing the Alternate key. A particular mnemonic character should only be used once within the set of menu items contained either in the same menu as the menu item or in the application's main menu.

Radio-Style Grouping

By using a few methods of the NSMenuItem protocol, you can implement radio-style groupings of menu commands. In other words, you can have a grouping of menu commands (usually segregated visually with separator items) and only one command in the group can be selected; the selected item is marked by an image, usually a radio-button image, but sometimes a checkmark. If the user selects another command in the group, the previous command is unmarked and the selected command displays the image. As an example of a radio-style grouping, a game could have three commands to indicate the level of play: Beginner, Intermediate, and Advanced.

To implement this feature, first set the images you want to use for the possible command states: "on," "off," and "mixed" (the last is useful for triple-state or indeterminate situations). To set the image, use the commands **setOnStateImage:**, **setOffStateImage:**, and **setMixedStateImage:**. The default image for the "on" state is a checkmark (NSMenuCheckmark) and for the "mixed" state the image is a dash (NSMenuMixedState). The "off" state typically has no image. The radio-button image (which you must set explicitly) is NSMenuRadio.

In an action method that responds to all commands in the group use **setState:** to uncheck the menu item that is currently marked:

```
[curItem setState:NSOffState];
```

Then mark the newly selected command:

```
[sender setState:NSOnState];
```

Method Types

Creating conforming NSMenuItem objects

– initWithTitle:action:keyEquivalent:

Enabling a menu item

– setEnabled:
– isEnabled

Setting the target and action

– setTarget:
– target
– setAction:
– action

Setting the title

– setTitle:
– title

Setting the tag

– setTag:
– tag

Setting the state

– setState:
– state

Setting the image

– setImage:
– image
– setOnStateImage:
– onStateImage
– setOffStateImage:
– offStateImage
– setMixedStateImage:
– mixedStateImage

Managing submenus

– setSubmenu:
– submenu
– hasSubmenu

Getting a separator item

+ separatorItem
– isSeparatorItem

Setting the owning menu

– setMenu:
– menu

Managing key equivalents

– setKeyEquivalent:
– keyEquivalent
– setKeyEquivalentModifierMask:
– keyEquivalentModifierMask

Managing mnemonics

– setMnemonicLocation:
– mnemonicLocation
– setTitleWithMnemonic:
– mnemonic

Managing user key equivalents

+ setUsesUserKeyEquivalents:
+ usesUserKeyEquivalents
– userKeyEquivalent

Representing an object

– setRepresentedObject:
– representedObject

Class Methods

separatorItem

+ (id <NSMenuItem>)**separatorItem**

Returns a menu item that is used to separate logical groups of menu commands. This menu item is disabled. The default separator item is a simple horizontal line.

See also: – **isSeparatorItem**, – **setEnabled:**

setUsesUserKeyEquivalents:

+ (void)**setUsesUserKeyEquivalents:**(BOOL)*flag*

If *flag* is YES, menu items conform to user preferences for key equivalents; otherwise, the key equivalents originally assigned to the menu items are used.

See also: **+ usesUserKeyEquivalents, – userKeyEquivalent**

usesUserKeyEquivalents

+ (BOOL)**usesUserKeyEquivalents**

Returns YES if menu items conform to user preferences for key equivalents; otherwise, returns NO.

See also: **+ setUsesUserKeyEquivalents:, – userKeyEquivalent**

Instance Methods

action

– (SEL)**action**

Returns the receiver's action method.

See also: **– target, – setAction:**

hasSubmenu

– (BOOL)**hasSubmenu**

Returns YES if the receiver has a submenu, NO if it doesn't.

See also: – **setSubmenu:forItem:**(NSMenu)

image

– (NSImage *)**image**

Returns the image displayed by the receiver, or **nil** if it displays no image.

See also: **– setImage:**

initWithTitle:action:keyEquivalent:

– (id)**initWithTitle:**(NSString *)*itemName*
action:(SEL)*anAction*
keyEquivalent:(NSString *)*charCode*

Returns an initialized instance of an object that conforms to the NSMenuItem protocol. The arguments *itemName* and *charCode* must not be **nil** (if there is no title or key equivalent, specify an empty NSString). The *anAction* argument must be a valid selector or NULL For instances of the NSMenItem class, the default initial state is NSStateOff, the default on-state image is a checkmark, and the default mixed-state image is a dash.

isEnabled

– (BOOL)**isEnabled**

Returns YES if the receiver is enabled, NO if not.

See also: – **setEnabled:**

isSeparatorItem

– (BOOL)**isSeparatorItem**

Returns whether the receiver is a separator item (that is, a menu item used to visually segregate related menu items).

See also: + **separatorItem**

keyEquivalent

– (NSString *)**keyEquivalent**

Returns the receiver's unmodified keyboard equivalent, or the empty string if one hasn't been defined. Use **keyEquivalentModifierMask** to determine the modifier mask for the key equivalent.

See also: **– userKeyEquivalent, – mnemonic, – setKeyEquivalent:**

keyEquivalentModifierMask

– (unsigned int)**keyEquivalentModifierMask**

Returns the receiver's keyboard equivalent modifier mask.

See also: **– setKeyEquivalentModifierMask:**

menu

– (NSMenu *)**menu**

Returns the menu to which the receiver belongs, or **nil** if no menu has been set.

See also: – **setMenu:**

mixedStateImage

– (NSImage *)**mixedStateImage**

Returns the image used to depict a “mixed state.” A mixed state is useful for indicating “off” and “on” attribute values in a group of selected objects, such as a selection of text containing bold and plain (non-bolded) worlds.

See also: – **setMixedStateImage:**

mnemonic

– (NSString *)mnemonic

Returns the character in the menu item title that appears underlined for use as a mnemonic. If there is no mnemonic character, returns an empty string.

See also: – **setTitleWithMnemonic:**

mnemonicLocation

– (unsigned int)**mnemonicLocation**

Returns the position of the underlined character in the menu item title used as a mnemonic. The position is the zero based index of that character in the title string. If the receiver has no mnemonic character, returns NSNotFound.

See also: – **setMnemonicLocation:**

offStateImage

– (NSImage *)offStateImage

Returns the image used to depict the receiver’s “off” state, or **nil** if the image has not been set.

See also: – **setOffStateImage:**

onStateImage

– (NSImage *)**onStateImage**

Returns the image used to depict the receiver's "on" state, or **nil** if the image has not been set.

See also: – **setOnStateImage:**

representedObject

– (id)**representedObject**

Returns the object that the receiving menu item represents. For example, you might have a menu list the names of views that are swapped into the same panel. The represented objects would be the appropriate NSView objects. The user would then be able to swtich back and forth between the different views that are displayed by selecting the various menu items.

See also: – **tag**, – **setRepresentedObject:**

setAction:

– (void)**setAction:**(SEL)*aSelector*

Sets the receiver's action method to *aSelector*.

See also: – **setTarget:**, – **action**

setEnabled:

– (void)**setEnabled:**(BOOL)*flag*

Sets whether the receiver is enabled based on *flag*. If a menu item is disabled, it's keyboard equivalent and mnemonic are also disabled. See the NSMenuValidation informal protocol specification for cautions regarding this method.

See also: – **isEnabled**

setImage:

– (void)**setImage:**(NSImage *)*menuImage*

Set's the receiver's image to *menuImage*. If *menuImage* is **nil**, the current image (if any) is removed. This image is not affected by changes in menu-item state.

See also: – **image**

setKeyEquivalent:

– (void)**setKeyEquivalent:**(NSString *)*aString*

Sets the receiver's unmodified key equivalent to *aString*. If you want to remove the key equivalent from a menu item, pass an empty string (@"") for *aString* (never pass **nil**). Use **setKeyEquivalentModifierMask:** to set the appropriate mask for the modifier keys for the key equivalent.

See also: **– setMnemonicLocation:, – keyEquivalent**

setKeyEquivalentModifierMask:

– (void)**setKeyEquivalentModifierMask:**(unsigned int)*mask*

Sets the receiver's keyboard equivalent modifiers (indicating modifiers such as the Shift or Alternate keys) to those in *mask*. *mask* is an integer bit field containing any of these modifier key masks, combined using the C bitwise OR operator:

NSShiftKeyMask
NSAlternateKeyMask
NSCommandKeyMask

On Mach, you should always set NSCommandKeyMask in *mask*; on Microsoft Windows, this is not required.

NSShiftKeyMask is relevant only for function keys; that is, for key events whose modifier flags include NSFunctionKeyMask. For all other key events NSShiftKeyMask is ignored and characters typed while the Shift key is pressed are interpreted as the shifted versions of those characters; for example, Command-Shift-'c' is interpreted as Command-'C'.

See the NSEvent class specification for more information about modifier mask values.

See also: **– keyEquivalentModifierMask**

setMenu:

– (void)**setMenu:**(NSMenu *)*aMenu*

Sets the receiver's menu to *aMenu*. This method is invoked by the owning NSMenu when the receiver is added or removed. You shouldn't have to invoke this method in your own code, although it can be overriden to provide specialized behavior.

See also: **– menu**

setMixedStateImage:

– (void)**setMixedStateImage:**(NSImage *)*itemImage*

Sets the image of the receiver that indicates a”mixed” state, that is, a state neither “on” or “off.” If *itemImage* is **nil**, any current mixed-state image is removed.

See also: – **mixedStateImage**, – **setOffStateImage:**, – **setOnStateImage:**, – **setState:**

setMnemonicLocation:

– (void)**setMnemonicLocation:**(unsigned int)*location*

Sets the character of the menu item title at *location* that is to be underlined. *location* must be between 0 and 254. This character identifies the access key on Windows by which users can access the menu item.

See also: – **mnemonicLocation**

setOffStateImage:

– (void)**setOffStateImage:**(NSImage *)*itemImage*

Sets the image of the receiver that indicates an “off” state. If *itemImage* is **nil**, any current off-state image is removed.

See also: – **offStateImage**, – **setMixedStateImage:**, – **setOnStateImage:**, – **setState:**

setOnStateImage:

– (void)**setOnStateImage:**(NSImage *)*itemImage*

Sets the image of the receiver that indicates an “on” state. If *itemImage* is **nil**, any current off-state image is removed.

See also: – **onStateImage**, – **setMixedStateImage:**, – **setOffStateImage:**, – **setState:**

setRepresentedObject:

– (void)**setRepresentedObject:**(id)*anObject*

Sets the object represented by the receiver to *anObject*. By setting a represented object for a menu item you make an association between the menu item and that object. The represented object functions as a more specific form of tag that allows you to associate any object, not just an **int**, with the items in a menu.

For example, an NSView object might be associated with a menu item—when the user chooses the menu item, the represented object is fetched and displayed in a panel. Several menu items might control the display of multiple views in the same panel.

See also: – **setTag:**, – **representedObject**

setState:

– (void)**setState:**(int)*itemState*

Sets the state of the receiver to *itemState*, which should be one of NSOffState, NSOnState, or NSMixedState. The image associated with the new state is displayed to the left of the menu item.

See also: – **state**, – **setMixedStateImage:**, – **setOffStateImage:**, – **setOnStateImage:**

setSubmenu:

– (void)**setSubmenu:**(NSMenu *)*aSubmenu*

Sets the submenu of the receiver to *aSubmenu*. The default implementation of the NSMenuItem class raises an exception if *aSubmenu* already has a supermenu.

See also: – **submenu**, – **hasSubmenu**

setTag:

– (void)**setTag:**(int)*anInt*

Sets the receiver's tag to *anInt*.

See also: – **setRepresentedObject:**, – **tag**

setTarget:

– (void)**setTarget:**(id)*anObject*

Sets the receiver's target to *anObject*.

See also: **– setAction:, – target**

setTitle:

– (void)**setTitle:**(NSString *)*aString*

Sets the receiver's title to *aString*.

See also: – **title**

setTitleWithMnemonic:

– (void)**setTitleWithMnemonic:**(NSString *)*aString*

Sets the title of a menu item with a character underlined to denote an access key (Windows only). Use an ampersand character to mark the character (the one following the ampersand) to be underlined. For example, the following message causes the 'c' in 'Receive' to be underlined:

```
[aMenuItem setTitleWithMnemonic:NSLocalizedString(@"Re&ceive")];
```

See also: – **mnemonic**, – **setMnemonicLocation:**

state

– (int)**state**

Returns the state of the receiver, which is NSOffState (the default), NSOnState, or NSMixedState.

See also: – **setState:**

submenu

– (NSMenu *)**submenu**

Returns the submenu associated with the receiving menu item, or **nil** if no submenu is associated with it. In the implementation of the NSMenuItem class, if the receiver responds YES to **hasSubmenu**, the submenu is returned.

See also: – **hasSubmenu**, – **setSubmenu**:

tag

– (int)**tag**

Returns the receiver's tag.

See also: – **representedObject**, – **setTag:**

target

– (id)**target**

Returns the receiver's target.

See also: **– action, – setTarget:**

title

– (NSString *)**title**

Returns the receiver's title.

See also: **– setTitle:**

userKeyEquivalent

– (NSString *)**userKeyEquivalent**

Returns the user-assigned key equivalent for the receiver.

See also: **– keyEquivalent**

NSMenuValidation

(informal protocol)

Category Of: NSObject

Declared In: AppKit/NSMenu.h

Protocol Description

This informal protocol allows your application to update the enabled or disabled status of an NSMenuItem. It declares only one method, **validateMenuItem:**. By default, every time a user event occurs, NSMenu automatically enables and disables each visible menu item based on criteria described in "Automatic Updating of NSMenuItems," below. Implement **validateMenuItem:** in cases where you want to override NSMenu's default enabling scheme.

NSMenuItems can be enabled or disabled in two ways: explicitly, by sending the **setEnabled:** message, or automatically, as described below. Automatic updating can be turned on and off with NSMenu's **setAutoenablesItems:** message.

Automatic Updating of NSMenuItems

Whenever a user event occurs, the NSMenu object updates the status of every one of its visible menu items. To update the status of a menu item, an NSMenu tries to find the object that responds to the NSMenuItem's action message. It searches the following objects in the given order until it finds one that responds to the action message (note that it doesn't actually send the action message):

- The NSMenuItem's target. If the target is non-**nil**, the search ends here whether the target responds or not.
- The key window's responder chain, starting with its first responder.
- The key window itself.
- The key window's delegate.
- The main window's responder chain, starting with its first responder.
- The main window itself.
- The main window's delegate.
- The NSApplication object.
- The NSApplication object's delegate.

If none of these objects responds to the action message, the menu item is disabled. If NSMenu finds an object that responds to the action message, it then checks to see if that object responds to the **validateMenuItem:** method (the method declared in this informal protocol). If **validateMenuItem:** is not

implemented in that object, the menu item is enabled. If it is implemented, the return value of **validateMenuItem:** indicates whether the menu item should be enabled or disabled.

Here is an example of using **validateMenuItem:** to override automatic enabling. If your application has a Copy menu item that sends the **copy:** action message to the first responder, that menu item is automatically enabled any time an object that responds to **copy:**, such as an NSText object, is the first responder of the key or main window. If you create a class whose instances might become the first responder, and which doesn't support copying of everything it allows the user to select, you should implement **validateMenuItem:** in that class. **validateMenuItem:** will then return NO if items that can't be copied are selected (or if no items are selected) and YES if all items in the selection can be copied. By implementing **validateMenuItem:**, you can have the Copy menu item disabled even though the target object does implement the **copy:** method. If a class never permits copying, then you simply omit an implemention of **copy:** in that class, and the Copy menu item is disabled automatically whenever an instance of that class is the first responder.

If you send a **setEnabled:** message to enable or disable a menu item when automatic updating is turned on (with NSMenu's **setAutoEnablesItems:**), other objects might undo what you have done after another user event occurs. Hence you can never be sure that the menu item will remain the way you set it. If your application must use **setEnabled:**, turn off the automatic enabling of menu items in order to get predictable results.

Instance Methods

validateMenuItem:

– (BOOL)**validateMenuItem:**(NSMenuItem *)*aMenuItem*

Implemented to override the default action of enabling or disabling *aMenuItem*. The object implementing this method must be the target of *aMenuItem*. It returns YES to enable the *aMenuItem*, NO to disable it. You can determine which menu item *aMenuItem* is by querying it for its title, tag, or action.

The following example beeps and disables the menu item "Next Record" if the selected line in a table view is the last one; conversely, it beeps and disables the menu item "Prior Record" if the selected row is the first one in the table view. (**countryKeys** is an array of names appearing in the table view.)

```
- (BOOL)validateMenuItem:(NSMenuItem *)anItem
{
    int row = [tableView selectedRow];
    if ([[anItem title] isEqualToString:@"Next Record"] &&
        (row == [countryKeys indexOfObject:[countryKeys lastObject]])) {
        return NO;
    }
    if ([[anItem title] isEqualToString:@"Prior Record"] && row == 0 ) {
        return NO;
    }
    return YES;
}
```

NSNibAwaking

(informal protocol)

Category Of: NSObject

Declared In: AppKit/NSNibLoading.h

Protocol Description

This informal protocol consists of a single method, **awakeFromNib**. Classes can implement this method to perform final initialization of state after objects have been loaded from an Interface Builder archive.

Instance Methods

awakeFromNib

– (void)**awakeFromNib**

Implemented to prepare the receiver for service after it has been loaded from an Interface Builder archive, or *nib file*. An **awakeFromNib** message is sent to each object loaded from the archive, but only if it can respond to the message, and only after all the objects in the archive have been loaded and initialized. When an object receives an **awakeFromNib** message, it's guaranteed to have all its outlet instance variables set.

Note: This method is also sent during Interface Builder's test mode to objects instantiated from loaded palettes, which include executable code for the objects. It isn't sent to objects defined solely by using the Classes display of the nib file window in Interface Builder.

When an Interface Builder archive is loaded into an application, each custom object from the archive is first initialized with an **init** message, or **initWithFrame:** if the object is a kind of NSView. It's then more specifically initialized with the properties that it was configured with using Interface Builder. This part of the initialization process uses any **set***Variable***:** methods that are available (where *variable* is the name of an instance variable whose value was set in Interface Builder). Finally, after all the objects are fully initialized, each receives an **awakeFromNib** message.

The order in which objects are loaded from the archive is not guaranteed. Therefore, it's possible for a **set***Variable***:** message to be sent to an object before its companion objects have been unarchived. For this reason, **set***Variable***:** methods should not send messages to other objects in the archive. However, messages to other objects can safely be sent from within **awakeFromNib**—by which time it's assured that all the objects are unarchived and initialized (though not necessarily awakened, of course).

Typically, **awakeFromNib** is implemented for classes whose instances are used as the owners of a loaded nib file (shown as "File's Owner" in Interface Builder). Such a class has the express purpose of connecting the loaded objects with objects in the application, and can thereafter be disposed of, or remain in the

capacity of a controller or coordinator for the loaded objects. For example, suppose that a nib file contains two custom views that must be positioned relative to each other at run time. Trying to position them when either one of the views is initialized (in **initWithCoder:** or a **set***Variable***:** method) might fail, since the other views might not be unarchived and initialized yet. However, it can be done in the nib file owner's **awakeFromNib** method (**firstView** and **secondView** are outlets of the file's owner):

```
- (void)awakeFromNib
{
    NSRect viewFrame;

    if ([[self superclass] instancesRespondToSelector:@selector(awakeFromNib)]) {
        [super awakeFromNib];
    }
    viewFrame = [firstView frame];
    viewFrame.origin.x += viewFrame.size.width;
    [secondView setFrame:viewFrame];
    return;
}
```

Note the testing of the superclass before invoking its implementation of **awakeFromNib**. The Application Kit declares a prototype for this method, but doesn't implement it. Because there's no default implementation of **awakeFromNib**, be sure to invoke it only when the object does in fact respond.

See also: + **loadNibNamed:owner:** (NSBundle Additions),
– **awakeAfterUsingCoder** (NSObject class of the Foundation Kit),
– **initWithCoder:** (NSCoding protocol of the Foundation Kit),
+ **initialize** (NSObject class of the Foundation Kit)

NSServicesRequests

(informal protocol)

Category Of: NSObject

Declared In: AppKit/NSApplication.h

Protocol Description

This informal protocol consists of two methods, **writeSelectionToPasteboard:types:** and **readSelectionFromPasteboard:**. The first is implemented to provide data to a remote service, and the second to receive any data the remote service might send back. Both respond to messages that are generated when the user chooses a command from the Services menu.

Instance Methods

readSelectionFromPasteboard:

- (BOOL)**readSelectionFromPasteboard:**(NSPasteboard *)*pboard*

Implemented to replace the current selection (that is, the text or objects that are currently selected) with data read from the Pasteboard object *pboard*. The data would have been placed in the pasteboard by another application in response to a remote message from the Services menu. A **readSelectionFromPasteboard:** message is sent to the same object that previously received a **writeSelectionToPasteboard:types:** message.

There's no default **readSelectionFromPasteboard:** method. The Application Kit declares a prototype for this method, but doesn't implement it.

writeSelectionToPasteboard:types:

- (BOOL)**writeSelectionToPasteboard:**(NSPasteboard *)*pboard* **types:**(NSArray *)*types*

Implemented to write the current selection to the Pasteboard object *pboard*. The selection should be written as one or more of the data types listed in *types*. After writing the data, this method should return YES. If for any reason it can't write the data, it should return NO.

A **writeSelectionToPasteboard:types:** message is sent to the first responder when the user chooses a command from the Services menu, but only if the receiver didn't return **nil** to a previous **validRequestorForSendType:returnType:** message.

After this method writes the data to the pasteboard, a remote message is sent to the application that provides the service the user requested. If the service provider supplies return data to replace the selection, the first responder will then receive a **readSelectionFromPasteboard:** message.

There's no default **writeSelectionToPasteboard:types:** method. The Application Kit declares a prototype for this method, but doesn't implement it.

See also: – **validRequestorForSendType:returnType:** (NSResponder class)

NSTableDataSource

(informal protocol)

Category Of: NSObject

Declared In: AppKit/NSTableView.h

Category Description

The NSTableDataSource category declares the methods that an NSTableView uses to access the contents of its data source object. It determines how many rows to display by sending a **numberOfRowsInTableView:** message, and accesses individual values with the **tableView: objectValueForTableColumn:row:** and **tableView:setObjectValue:forTableColumn:row:** methods. A data source must implement the first two methods to work with an NSTableView, but if it doesn't implement the third the NSTableView simply provides read-only access to its contents.

The NSTableView treats objects provided by its data source as values to be displayed in NSCell objects. If these objects aren't of common value classes—such as NSString, NSNumber, and so on—you'll need to create a custom NSFormatter to display them. See the NSFormatter class specification for more information.

Suppose that an NSTableView's column identifiers are set up as NSStrings containing the names of attributes for the column, such as "Last Name", "City", and so on, and that the data source stores its records as an NSMutableArray, called **records**, of NSMutableDictionary objects using those names as keys. Here's a small example, given as an ASCII property list:

```
(
    {
        "Last Name" = Anderson;
        "First Name" = James;
        Abode = apartment;
        City = "San Francisco";
    },
    {
        "Last Name" = Beresford;
        "First Name" = Keith;
        Abode = apartment;
        City = "Redwood City";
    }
)
```

With such a record structure, this implementation of **tableView:objectValueForTableColumn:row:** suffices to retrieve values for the NSTableView:

```
- (id)tableView:(NSTableView *)aTableView
    objectValueForTableColumn:(NSTableColumn *)aTableColumn
    row:(int)rowIndex
{
    id theRecord, theValue;

    NSParameterAssert(rowIndex >= 0 && rowIndex < [records count]);
    theRecord = [records objectAtIndex:rowIndex];
    theValue = [theRecord objectForKey:[aTableColumn identifier]];
    return theValue;
}
```

Here's the corresponding method for setting values:

```
- (void)tableView:(NSTableView *)aTableView
    setObjectValue:anObject
    forTableColumn:(NSTableColumn *)aTableColumn
    row:(int)rowIndex
{
    id theRecord;

    NSParameterAssert(rowIndex >= 0 && rowIndex < [records count]);
    theRecord = [records objectAtIndex:rowIndex];
    [theRecord setObject:anObject forKey:[aTableColumn identifier]];
    return;
}
```

Finally, **numberOfRowsInTableView:** simply returns the count of the NSArray:

```
- (int)numberOfRowsInTableView:(NSTableView *)aTableView
{
    return [records count];
}
```

In each case, the NSTableView that sends the message is provided as *aTableView*. A data source object that manages several sets of data can choose the appropriate set based on which NSTableView sends the message.

Method Types

Getting values

– numberOfRowsInTableView:
– tableView:objectValueForTableColumn:row:

Setting values

– tableView:setObjectValue:forTableColumn:row:

Instance Methods

numberOfRowsInTableView:

– (int)**numberOfRowsInTableView:**(NSTableView *)*aTableView*

Returns the number of records managed for *aTableView* by the data source object. An NSTableView uses this method to determine how many rows it should create and display.

tableView:objectValueForTableColumn:row:

– (id)**tableView:**(NSTableView *)*aTableView*
objectValueForTableColumn:(NSTableColumn *)*aTableColumn*
row:(int)*rowIndex*

Returns an attribute value for the record in *aTableView* at *rowIndex*. *aTableColumn* contains the identifier for the attribute, which you get by using NSTableColumn's **identifier** method. For example, if *aTableColumn* stands for the city that an employee lives in and *rowIndex* specifies the record for an employee who lives in Portland, this method returns an object with a string value of "Portland". See the category description for an example.

tableView:setObjectValue:forTableColumn:row:

– (void)**tableView:**(NSTableView *)*aTableView*
setObjectValue:(id)*anObject*
forTableColumn:(NSTableColumn *)*aTableColumn*
row:(int)*rowIndex*

Sets an attribute value for the record in *aTableView* at *rowIndex*. *anObject* is the new value, and *aTableColumn* contains the identifier for the attribute, which you get by using NSTableColumn's **identifier** method. See the category description for an example.

NSTextAttachmentCell

Adopted By: NSTextAttachmentCell

Declared In: AppKit/NSTextAttachment.h

Protocol Description

The NSTextAttachmentCell protocol declares the interface for objects that draw text attachment icons and handle mouse events on their icons. With the exceptions of **cellBaselineOffset:**, **setAttachment:** and **attachment**, all of these methods are implemented by the NSCell class and described in that class specification.

See the NSAttributedString and NSTextView class specifications for general information on text attachments.

Method Types

Drawing

– drawWithFrame:inView:
– highlight:withFrame:inView:

Cell size and position

– cellSize
– cellBaselineOffset

Event handling

– wantsToTrackMouse
– trackMouse:inRect:ofView:untilMouseUp:

Setting the attachment

– setAttachment:
– attachment

Instance Methods

attachment

– (NSTextAttachment *)**attachment**

Returns the text attachment object that owns the receiver.

See also: – **setAttachment:**

cellBaselineOffset

– (NSPoint)**cellBaselineOffset**

Returns the position where the attachment cell's image should be drawn in text, relative to the current point established in the glyph layout. The image should be drawn so that its lower left corner lies on this point.

See also: – **icon** (NSFileWrapper)

cellSize

– (NSSize)**cellSize**

Returns the size of the attachment's icon.

See also: – **icon** (NSFileWrapper), – **fileWrapper** (NSTextAttachment)

drawWithFrame:inView:

– (void)**drawWithFrame:**(NSRect)*cellFrame* **inView:**(NSView *)*aView*

Draws the receiver's image within *cellFrame* in *aView*, which should be the focus view.

See also: – **drawWithFrame:inView:** (NSCell), – **lockFocus** (NSView)

highlight:withFrame:inView:

– (void)**highlight:**(BOOL)*flag*
withFrame:(NSRect)*cellFrame*
inView:(NSView *)*aView*

Draws the receiver's image—with highlighting if *flag* is YES—within *cellFrame* in *aView*, which should be the focus view.

See also: – **highlight:withFrame:inView:** (NSCell), – **lockFocus** (NSView)

setAttachment:

– (void)**setAttachment:**(NSTextAttachment *)*anAttachment*

Sets the text attachment object that owns the receiver to *anAttachment*, without retaining it (the text attachment, as the owner, retains the cell).

See also: – **attachment**, – **setAttachmentCell:** (NSTextAttachment)

trackMouse:inRect:ofView:untilMouseUp:

– (BOOL)**trackMouse:**(NSEvent *)*theEvent*
inRect:(NSRect)*cellFrame*
ofView:(NSView *)*aTextView*
untilMouseUp:(BOOL)*flag*

Handles a mouse-down event on the receiver's image. *theEvent* is the mouse-down event. *cellFrame* is the region of *aTextView* in which further mouse events should be tracked. *aTextView* is the view which received the event. It's assumed to be an NSTextView, and should be the focus view. If *flag* is YES, the receiver tracks the mouse until a mouse-up event occurs; if *flag* is NO, it stops tracking when a mouse-dragged event occurs outside of *cellFrame*. Returns YES if the receiver successfully finished tracking the mouse (typically through a mouse-up event), NO otherwise (such as when the mouse is dragged outside *cellFrame*).

NSTextAttachmentCell's implementation of this method calls upon *aTextView*'s delegate to handle the event. If *theEvent* is a mouse-up event for a double click, the text attachment cell sends the delegate a **textView:doubleClickedOnCell:inRect:** message and returns YES. Otherwise, depending on whether the user clicks or drags the cell, it sends the delegate a **textView:clickedOnCell:inRect:** or a **textView: draggingCell:inRect:event:** message and returns YES. NSTextAttachmentCell's implementation returns NO only if *flag* is NO and the mouse is dragged outside of *cellFrame*. The delegate methods are invoked only if the delegate responds.

See also: – **wantsToTrackMouse**, – **trackMouse:inRect:ofView:untilMouseUp:** (NSCell),
– **lockFocus** (NSView)

wantsToTrackMouse

– (BOOL)**wantsToTrackMouse**

Returns YES if the receiver will handle a mouse event occurring over its image (to support dragging, for example), NO otherwise. NSTextAttachmentCell's implementation of this method returns YES. The NSView containing the cell should invoke this method before sending a **trackMouse:inRect:ofView: untilMouseUp:** message.

For an attachment in an attributed string, if the attachment cell returns NO its attachment character should be selected rather than the cell being asked to track the mouse. this results in the attachment icon behaving as any regular glyph in text.

NSTextInput

Adopted By: NSInputManager

Declared In: AppKit/NSInputManager.h

Protocol Description

Note: *This class specification is incomplete and has not received a technical review. It is included in this release to test the linkage between the application development tools and the on-line documentation. What information it contains should be considered preliminary and subject to change.*

The methods of the NSTextInput protocol are implemented by objects—for example, responders—that handle text input. A client object such as a text view ("the client object") that speaks this protocol must be reponsible for the following things:

1. The client object must maintain a "marked region" within which text input and, possibly, character conversion take place. The marked region may have a length of zero. The client object must maintain an "insertion point"—typically at the end of the marked region, though it may be within the region. The "selection" within the client object, if any, is entirely contained within the marked region whenever there is a marked region.2. The client object is responsible for sending messages to currentInputManager when the mouse goes down inside the marked region, or when the mouse leaves the marked region. Within the marked region, this allows the selection to be changed. Out of the marked region, it allows the region to be "abandoned". (See below.)

3. When there is a non-zero marked region, the client object is responsible for notifying the input manager when the selection changes, or when other programmatic changes to the text affect the marked region. It can do this by sending a message to set the selection, abandon the marked region, etc.

4. When the client object relinquishes first responder, it will typically send markedTextWillBeAbandoned: to the currentInputManager. It must send markedTextWillBeAbandoned: when its insertion point (or selection moves outside the marked range. The server will typically respond by simply unmarking the region, but may remove the marked region's text entirely.

Note: If this protocol is not implemented by a client object that does have a keyDown: method, then in-line input is not possible for that client object, and will have to be handled externally.

The NSTextInput protocol is implemented by a IM to receive input from the server on behalf of the current client, and otherwise mediate between the client object and the server. It then forwards the corresponding messages to the client, or gets information from the client to pass back to the server, as appropriate.

The message passing between NSApp, IM, UIobj, and Server is all synchronous. That is, e.g., when IM sends a message to Server, any reply comes back and is relayed to UIobj before the original message returns.

A key binding manager splits the stream of keyDown: messages (intercepted by NSResopnder) into commands and text. If there is an Input Manager in the loop, it will further use any of these NSTextInput messages to control the marked region.

Method Types

Marking text

– setMarkedText:selectedRange:
– getMarkedText:selectedRange:
– hasMarkedText
– unmarkText

Other

– conversationIdentifier
– doCommandBySelector:
– insertText:

Instance Methods

conversationIdentifier

– (long)**conversationIdentifier**

Returns a number used to identify the receiver's input management session to the input server.

<<more information forthcoming>>

doCommandBySelector:

– (void)**doCommandBySelector:**(SEL)*aSelector*

Attempts to invoke *aSelector* or pass the message up the responer chain. This method is invoked by an input manager in response to an **interpretKeyEvents:** message.

<<more information forthcoming>>

See also: – **interpretKeyEvents:** (NSResponder)

getMarkedText:selectedRange:

– (void)**getMarkedText:**(out NSString **)*aString* **selectedRange:**(out NSRange *)*aRange*

Returns by reference in *aString* the receiver's marked text, if any, and in *aRange* the range of the selection within *aString* (not in terms of the receiver's entire text stream).

hasMarkedText

– (BOOL)**hasMarkedText**

Returns YES if the receiver has text that's still being interpreted by the input manager, NO if it doesn't.

<<more information forthcoming>>

insertText:

– (void)**insertText:**(NSString *)*aString*

Inserts *aString* into the receiver's text stream. This method is invoked by an input manager in response to an **interpretKeyEvents:** message.

<<more information forthcoming>>

See also: – **interpretKeyEvents:** (NSResponder)

setMarkedText:selectedRange:

– (void)**setMarkedText:**(NSString *)*aString* **selectedRange:**(NSRange)*selRange*

<<forthcoming>>

unmarkText

– (void)**unmarkText**

Removes any marking from pending input text, and accepts the text in its current state.

<<more information forthcoming>>

Defined Types

NSBorderType

DECLARED IN AppKit/NSView.h

SYNOPSIS

typedef enum _NSBorderType {
    NSNoBorder,
    NSLineBorder,
    NSBezelBorder,
    NSGrooveBorder
} **NSBorderType**;

DESCRIPTION This type represents the kinds of border that can be drawn around certain NSView subclasses.

NSButtonType

DECLARED IN AppKit/NSButtonCell.h

SYNOPSIS

typedef enum _NSButtonType {
    NSMomentaryPushButton,
    NSPushOnPushOffButton,
    NSToggleButton,
    NSSwitchButton,
    NSRadioButton,
    NSMomentaryChangeButton,
    NSOnOffButton,
    NSMomentaryLight,
} **NSButtonType**;

DESCRIPTION This type represents the way NSButtons and NSButtonCells behave when pressed, and the way they display their state. See NSButton's and NSButtonCell's **setButtonType:** methods for more information.

NSCellAttribute

DECLARED IN AppKit/NSCell.h

SYNOPSIS

```
typedef enum _NSCellAttribute {
    NSCellDisabled,
    NSCellState,
    NSPushInCell,
    NSCellEditable,
    NSChangeGrayCell,
    NSCellHighlighted,
    NSCellLightsByContents,
    NSCellLightsByGray,
    NSChangeBackgroundCell,
    NSCellLightsByBackground,
    NSCellIsBordered,
    NSCellHasOverlappingImage,
    NSCellHasImageHorizontal,
    NSCellHasImageOnLeftOrBottom,
    NSCellChangesContents,
    NSCellIsInsetButton,
    NSCellAllowsMixedState
} NSCellAttribute;
```

DESCRIPTION This is the type of the first argument to the NSCell methods **setCellAttribute:to:** and **cellAttribute:** methods. Some of the values apply not to NSCell but to one of its subclasses.

Often it's preferable to change cell attributes using more specialized methods like **setState:** or **setEditable:**.

NSCellImagePosition

DECLARED IN AppKit/NSCell.h

SYNOPSIS

typedef enum _NSCellImagePosition {
    NSNoImage,
    NSImageOnly,
    NSImageLeft,
    NSImageRight,
    NSImageBelow,
    NSImageAbove,
    NSImageOverlaps,
} **NSCellImagePosition**;

DESCRIPTION These constants represent the position of an NSButtonCell's NSImage relative to its title. See NSButton's and NSButtonCell's **setImagePosition:** and **imagePosition** methods for more information.

NSCellType

DECLARED IN AppKit/NSCell.h

SYNOPSIS

typedef enum _NSCellType {
    NSNullCellType,
    NSTextCellType,
    NSImageCellType,
} **NSCellType**;

DESCRIPTION The values of this type determine what kind of data an NSCell displays. NSCells of type NSTextCellType and NSImageCellType display text and images, respectively. NSCells of type NSNullCellType display nothing. See the NSCell methods **type** and **setType:** for more information.

NSDataLinkDisposition

DECLARED IN AppKit/NSDataLink.h

SYNOPSIS

```
typedef enum _NSDataLinkDisposition {
    NSLinkInDestination,
    NSLinkInSource,
    NSLinkBroken
} NSDataLinkDisposition;
```

DESCRIPTION Returned by NSDataLink's **disposition** method to identify a link as a destination link, a source link, or a broken link. See the NSDataLink class specification for more information on the dispositions of links.

NSDataLinkNumber

DECLARED IN AppKit/NSDataLink.h

SYNOPSIS

```
typedef int NSDataLinkNumber;
```

DESCRIPTION The type returned by NSDataLink's **linkNumber** method as a persistent identifier of a destination link.

NSDataLinkUpdateMode

DECLARED IN AppKit/NSDataLink.h

SYNOPSIS

```
typedef enum _NSDataLinkUpdateMode {
    NSUpdateContinuously,
    NSUpdateWhenSourceSaved,
    NSUpdateManually,
    NSUpdateNever
} NSDataLinkUpdateMode;
```

DESCRIPTION Used by NSDataLink's **setUpdateMode:** and **updateMode** methods to identify when a link's data is to be updated.

NSEventType

DECLARED IN AppKit/NSEvent.h

SYNOPSIS

```
typedef enum _NSEventType {
        NSLeftMouseDown,
        NSLeftMouseUp,
        NSRightMouseDown,
        NSRightMouseUp,
        NSMouseMoved,
        NSLeftMouseDragged,
        NSRightMouseDragged,
        NSMouseEntered,
        NSMouseExited,
        NSKeyDown,
        NSKeyUp,
        NSFlagsChanged,
        NSAppKitDefined,
        NSSystemDefined,
        NSApplicationDefined
        NSPeriodic,
        NSCursorUpdate
} NSEventType;
```

DESCRIPTION This type represents various kinds of events. It is the return type of NSEvent's type method, and the type of the first argument to NSEvent's **...EventWithType:** methods.

NSFontAction

DECLARED IN AppKit/NSFontManager.h

SYNOPSIS

typedef enum _NSFontAction {
    NSNoFontChangeAction,
    NSViaPanelFontAction,
    NSAddTraitFontAction,
    NSSizeUpFontAction,
    NSSizeDownFontAction,
    NSHeavierFontAction,
    NSLighterFontAction,
    NSRemoveTraitFontAction
} **NSFontAction**;

DESCRIPTION Values of this type tag the actions of font menu cells. When a font menu cell sends a message to NSFontManager, NSFontManager checks the cell for one of these tags.

This type is in the API for explanatory purposes only. You will never use it directly.

NSFontTraitMask

DECLARED IN AppKit/NSFontManager.h

SYNOPSIS typedef unsigned int **NSFontTraitMask;**

DESCRIPTION An NSFontTraitMask characterizes one or more of a font's traits. It's used as an argument type for NSAttributedString's **applyFontTraits:range:** method, NSCStringText's **setSelFontStyle:** method, and several of the methods in the NSFontManager class.

NSGlyph

DECLARED IN AppKit/NSFont.h

SYNOPSIS typedef unsigned int **NSGlyph**;

DESCRIPTION This type is used to specify PostScript glyphs in such NSFont methods as **glyphWithName:**.

NSGlyphInscription

DECLARED IN AppKit/NSLayoutManager.h

SYNOPSIS

```
typedef enum {
    NSGlyphInscribeBase,
    NSGlyphInscribeBelow,
    NSGlyphInscribeAbove,
    NSGlyphInscribeOverstrike,
    NSGlyphInscribeOverBelow
} NSGlyphInscription;
```

DESCRIPTION The inscribe attribute of an glyph determines how it is laid out relative to the previous glyph.

NSGlyphRelation

DECLARED IN AppKit/NSFont.h

SYNOPSIS

```
typedef enum _NSGlyphRelation {
    NSGlyphBelow,
    NSGlyphAbove,
} NSGlyphRelation;
```

DESCRIPTION This type specifies the position of a glyph in relation to the base glyph. Parameters of this type are used in the second slot of the NSFont method **positionOfGlyph:withRelation: toBaseGlyph:...**.

NSGradientType

DECLARED IN AppKit/NSButtonCell.h

SYNOPSIS

typedef enum _NSGradientType {
    NSGradientNone,
    NSGradientConcaveWeak,
    NSGradientConcaveStrong,
    NSGradientConvexWeak,
    NSGradientConvexStrong
} **NSGradientType**;

DESCRIPTION This type represents the darkness gradient of an NSButtonCell. A concave gradient is darkest in the top left corner, a convex gradient is darkest in the bottom right corner. A weak gradient has only weak darkness contrast between opposite corners; a strong gradient has strong contrast. See the NSButtonCell methods **gradient** and **setGradient:** for more information.

NSImageAlignment

DECLARED IN AppKit/NSImageCell.h

SYNOPSIS

typedef enum {
    NSImageAlignCenter,
    NSImageAlignTop,
    NSImageAlignTopLeft,
    NSImageAlignTopRight,
    NSImageAlignLeft,
    NSImageAlignBottom,
    NSImageAlignBottomLeft,
    NSImageAlignBottomRight,
    NSImageAlignRight
} **NSImageAlignment**;

DESCRIPTION This type defines the ways of aligning an NSImage within an NSImageCell. It is the return type for NSImageCell's and NSImageView's **imageAlignment** methods, and an argument type for their **setImageAlignment:** methods.

NSImageFrameStyle

DECLARED IN AppKit/NSImageCell.h

SYNOPSIS

```
typedef enum {
    NSImageFrameNone,
    NSImageFramePhoto,
    NSImageFrameGrayBezel,
    NSImageFrameGroove,
    NSImageFrameButton
} NSImageFrameStyle;
```

DESCRIPTION This type defines the kinds of frames that can appear around an NSImageCell. It is the return type for NSImageCell's and NSImageView's **imageFrameStyle** methods, and an argument type for their **setImageFrameStyle:** methods.

NSImageScaling

DECLARED IN AppKit/NSImageCell.h

SYNOPSIS

```
typedef enum {
    NSScaleProportionally,
    NSScaleToFit,
    NSScaleNone
} NSImageScaling;
```

DESCRIPTION This type defines the ways that an image can be scaled to fit an NSImageCell. The value NSScaleProportionally means that the image should be scaled in a way that preserves its proportions. The value NSScaleToFit means that the image should fit the NSView, even if that means its proportions must be distorted. The value NSScaleNone means that the image's size should be preserved, even if it must be clipped to fit the NSView.

NSInterfaceStyle

DECLARED IN AppKit/NSInterfaceStyle.h

SYNOPSIS

```
typedef enum {
    NSNoInterfaceStyle,
    NSNextStepInterfaceStyle,
    NSWindows95InterfaceStyle,
    NSMacintoshInterfaceStyle
} NSInterfaceStyle;
```

DECLARED IN This type defines the style of an application's user interface. It is returned by the **interfaceStyle** method and taken as an argument by the **setInterfaceStyle:** method. Both of the these methods are in the NSInterfaceStyle category of NSResponder.

For more information, see the function **NSInterfaceStyleForKey**.

NSLineBreakMode

DECLARED IN AppKit/NSParagraphStyle.h

SYNOPSIS

```
typedef enum _NSLineBreakMode {
    NSLineBreakByWordWrapping,
    NSLineBreakByCharWrapping,
    NSLineBreakByClipping,
    NSLineBreakByTruncatingHead,
    NSLineBreakByTruncatingTail,
    NSLineBreakByTruncatingMiddle
} NSLineBreakMode;
```

DESCRIPTION This type defines the ways that a long paragraph can be broken into lines. The possible values are described below.

Value	Meaning
NSLineBreakByWordWrapping	The default value. At the last possible word boundary, the paragraph wraps to the next line.

Value	Meaning
NSLineBreakByCharWrapping	At the last possible character, the paragraph wraps to the next line.
NSLineBreakByClipping	As much of the paragraph appears as will fit on a single line. This value has the same effect as NSLineBreakByTruncatingTail
NSLineBreakByTruncatingHead	As much of the paragraph appears as will fit on a single line. Characters from the start of the paragraph do not appear.
NSLineBreakByTruncatingTail	As much of the paragraph appears as will fit on a single line. Characters from the end of the paragraph do not appear.
NSLineBreakByTruncatingMiddle	As much of the paragraph appears as will fit on a single line. Characters from the middle of the paragraph do not appear.

NSLineMovementDirection

DECLARED IN AppKit/NSTextContainer.h

SYNOPSIS

```
typedef enum {
    NSLineDoesntMove,
    NSLineMovesLeft,
    NSLineMovesRight,
    NSLineMovesDown,
    NSLineMovesUp
} NSLineMovementDirection;
```

DESCRIPTION This is an argument type for the NSTextContainer method **lineFragmentRectForProposedRect: sweepDirection:movementDirection:remainingRect:**.

NSLineSweepDirection

DECLARED IN AppKit/NSTextContainer.h

SYNOPSIS

typedef enum {
    NSLineSweepLeft,
    NSLineSweepRight,
    NSLineSweepDown,
    NSLineSweepUp
} **NSLineSweepDirection**;

DESCRIPTION This is an argument type for the NSTextContainer method **lineFragmentRectForProposedRect: sweepDirection:movementDirection:remainingRect:**.

NSMatrixMode

DECLARED IN AppKit/NSMatrix.h

SYNOPSIS

typedef enum _NSMatrixMode {
    NSRadioModeMatrix,
    NSHighlightModeMatrix,
    NSListModeMatrix,
    NSTrackModeMatrix
} **NSMatrixMode**;

DESCRIPTION These constants represent the modes of operation of an NSMatrix, as described in the NSMatrix class specification.

NSModalSession

DECLARED IN AppKit/NSApplication.h

SYNOPSIS

typedef struct _NSModalSession ***NSModalSession**;

DESCRIPTION Variables of type NSModalSession point to information used by the system between **beginModalSession:for:** and **endModalSession:** messages.

NSPrinterTableStatus

DECLARED IN AppKit/NSPrinter.h

SYNOPSIS

typedef enum _NSPrinterTableStatus {
    NSPrinterTableOK,
    NSPrinterTableNotFound,
    NSPrinterTableError
} **NSPrinterTableStatus**;

DESCRIPTION These constants are used to describe the state of a printer-information table stored by an NSPrinter object. See the NSPrinter method **statusForTable:** for more information.

NSPrintingOrientation

DECLARED IN AppKit/NSPrintInfo.h

SYNOPSIS

typedef enum _NSPrintingOrientation {
    NSPortraitOrientation,
    NSLandscapeOrientation
} **NSPrintingOrientation**;

DESCRIPTION These constants represent the way a page is oriented for printing. In NSPortraitOrientation, the page is taller than it is wide; in NSLandscapeOrientation, the page is wider than it is tall. See the NSPrintInfo methods **orientation** and **setOrientation:** for more information.

NSPrintingPageOrder

DECLARED IN AppKit/NSPrintOperation.h

SYNOPSIS

typedef enum _NSPrintingPageOrder {
    NSDescendingPageOrder,
    NSSpecialPageOrder,
    NSAscendingPageOrder,
    NSUnknownPageOrder
} **NSPrintingPageOrder**;

DESCRIPTION This type represents the order in which pages are to be printed. The value NSSpecialPageOrder tells the spooler to not rearrange the pages. The value NSUnknownPageOrder means that no page order is written out. See the NSPrintOperation methods **pageOrder** and **setPageOrder:** for more information.

NSPrintingPaginationMode

DECLARED IN AppKit/NSPrintInfo.h

SYNOPSIS

typedef enum _NSPrintingPaginationMode {
    NSAutoPagination,
    NSFitPagination,
    NSClipPagination
} **NSPrintingPaginationMode**;

DESCRIPTION These constants represent the different ways in which an image is divided into pages. The value NSFitPagination forces the image to fit on one page. The value NSClipPagination allows the image to be clipped by the page.See the NSPrintInfo class specification for a fuller explanation.

NSRulerOrientation

DECLARED IN AppKit/NSRulerView.h

SYNOPSIS

```
typedef enum {
    NSHorizontalRuler,
    NSVerticalRuler
} NSRulerOrientation;
```

DESCRIPTION This type defines whether an NSRulerView will be displayed horizontally or vertically. It is the return type of NSRulerView's **orientation** method, and an argument to the NSRulerView methods **setOrientation:** and **initWithScrollView:orientation:**.

NSScrollArrowPosition

DECLARED IN AppKit/NSScroller.h

SYNOPSIS

```
typedef enum _NSScrollArrowPosition {
    NSScrollerArrowsMaxEnd,
    NSScrollerArrowsMinEnd,
    NSScrollerArrowsNone
} NSScrollArrowPosition;
```

DESCRIPTION These constants are used in NSScroller's **setArrowsPosition:** method to set the position of the arrows within the scroller.

NSScrollerArrow

DECLARED IN AppKit/NSScroller.h

SYNOPSIS

```
typedef enum _NSScrollerArrow {
    NSScrollerIncrementArrow,
    NSScrollerDecrementArrow
} NSScrollerArrow;
```

DESCRIPTION This is the type of the first argument to the NSScroller method **drawArrow:highlight:**. The value determines which scroll button is drawn.

NSScrollerPart

DECLARED IN AppKit/NSScroller.h

SYNOPSIS

```
typedef enum _NSScrollerPart {
    NSScrollerNoPart,
    NSScrollerDecrementPage,
    NSScrollerKnob,
    NSScrollerIncrementPage,
    NSScrollerDecrementLine,
    NSScrollerIncrementLine,
    NSScrollerKnobSlot
} NSScrollerPart;
```

DESCRIPTION These constants are used in Scroller's **hitPart** method to identify the part of the Scroller specified in a mouse event.

NSSelectionAffinity

DECLARED IN AppKit/NSTextView.h

SYNOPSIS

```
typedef enum _NSSelectionAffinity {
    NSSelectionAffinityUpstream,
    NSSelectionAffinityDownstream
} NSSelectionAffinity;
```

DESCRIPTION This is the return type of the NSTextView method **selectionAffinity**, and the type of the second argument to the NSTextView method **setSelectedRange:affinity:stillSelecting:**.

NSSelectionDirection

DECLARED IN AppKit/NSWindow.h

SYNOPSIS

typedef enum _NSSelectionDirection {
    NSDirectSelection,
    NSSelectingNext,
    NSSelectingPrevious
} **NSSelectionDirection**;

DESCRIPTION This is the return type of the NSWindow method **keyViewSelectionDirection**.

NSSelectionGranularity

DECLARED IN AppKit/NSTextView.h

SYNOPSIS

typedef enum _NSSelectionGranularity {
    NSSelectByCharacter,
    NSSelectByWord,
    NSSelectByParagraph
} **NSSelectionGranularity**;

DESCRIPTION This is the return type of the NSTextView method selectionGranularity, and the type of arguments to two other NSTextView methods, **setSelectionGranularity:** and **selectionRangeForProposedRange:granularity:**.

NSTextAlignment

DECLARED IN AppKit/NSText.h

SYNOPSIS

typedef enum _NSTextAlignment {
    NSLeftTextAlignment,
    NSRightTextAlignment,
    NSCenterTextAlignment,
    NSJustifiedTextAlignment,
    NSNaturalTextAlignment
} **NSTextAlignment**;

DESCRIPTION Variables of this type are used as arguments and return values for methods that specify text alignment.

NSTextTabType

DECLARED IN AppKit/NSParagraphStyle.h

SYNOPSIS

typedef enum _NSTextTabType {
    NSLeftTabStopType,
    NSRightTabStopType,
    NSCenterTabStopType,
    NSDecimalTabStopType
} **NSTextTabType**;

DESCRIPTION This is the return type of NSTextTab's **tabStopType** method, and an argument to NSTextTab's **initWithType:location:** method.

NSTIFFCompression

SYNOPSIS AppKit/NSBitmapImageRep.h

SYNOPSIS

typedef enum _NSTIFFCompression {
NSTIFFCompressionNone,
NSTIFFCompressionCCITTFAX3,
NSTIFFCompressionCCITTFAX4,
NSTIFFCompressionLZW,
NSTIFFCompressionJPEG,
NSTIFFCompressionNEXT,
NSTIFFCompressionPackBits,
NSTIFFCompressionOldJPEG
} **NSTIFFCompression**;

DESCRIPTION

These constants represent the various TIFF (*tag image file format*) data compression schemes. See the NSBitmapImageRep class specification for their meanings.

NSTitlePosition

DECLARED IN

AppKit/NSBox.h

SYNOPSIS

typedef enum _NSTitlePosition {
NSNoTitle,
NSAboveTop,
NSAtTop,
NSBelowTop,
NSAboveBottom,
NSAtBottom,
NSBelowBottom
} **NSTitlePosition**;

DESCRIPTION

This type represents the locations where an NSBox's title can be placed with respect to its border. Thus, for example, NSAboveTop means the title is above the top of the border, NSAtTop means the title breaks the top border, and so on. See the NSBox methods **titlePosition** and **setTitlePosition:**.

NSTrackingRectTag

DECLARED IN AppKit/NSView.h

SYNOPSIS typedef int **NSTrackingRectTag**;

DESCRIPTION This type describes the rectangle used to track the mouse. See the NSView methods **addTrackingRect:...** and **removeTrackingRect:**.

NSUsableScrollerParts

DECLARED IN AppKit/NSScroller.h

SYNOPSIS

```
typedef enum _NSUsableScrollerParts {
    NSNoScrollerParts,
    NSOnlyScrollerArrows,
    NSAllScrollerParts
} NSUsableScrollerParts;
```

DESCRIPTION This type defines the usable parts of an NSScroller; see the class specification for more information.

NSWindowDepth

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS typedef int **NSWindowDepth**;

DESCRIPTION This type represents the depth, or amount of memory, devoted to a single pixel in a window or screen.

Enumerations

NSApplication—Modal Session Return Values

DECLARED IN AppKit/NSApplication.h

SYNOPSIS

```
enum {
    NSRunStoppedResponse,
    NSRunAbortedResponse,
    NSRunContinuesResponse
};
```

DESCRIPTION Return values for **the NSApplication methods runModalFor:** and **runModalSession:**.

NSAttributedString—Underlining

DECLARED IN AppKit/NSAttributedString.h

SYNOPSIS

```
enum {
    NSSingleUnderlineStyle
};
```

DESCRIPTION This defines the only currently supported value for NSUnderlineStyleAttributeName.

NSButtonCell—State Masks

DECLARED IN AppKit/NSCell.h

SYNOPSIS

```
enum {
    NSNoCellMask,
    NSContentsCellMask,
    NSPushInCellMask,
    NSChangeGrayCellMask,
    NSChangeBackgroundCellMask
};
```

DESCRIPTION These masks are passed to the NSButtonCell methods **highlightsBy:** and **showsStateBy:**.

NSCell—Action Flags

DECLARED IN AppKit/NSEvent.h

SYNOPSIS

```
enum {
    NSLeftMouseDownMask,
    NSLeftMouseUpMask,
    NSRightMouseDownMask,
    NSRightMouseUpMask,
    NSMouseMovedMask,
    NSLeftMouseDraggedMask,
    NSRightMouseDraggedMask,
    NSMouseEnteredMask,
    NSMouseExitedMask,
    NSKeyDownMask,
    NSKeyUpMask,
    NSFlagsChangedMask,
    NSAppKitDefinedMask,
    NSSystemDefinedMask,
    NSApplicationDefinedMask,
    NSPeriodicMask,
    NSCursorUpdateMask,
    NSAnyEventMask
};
```

DESCRIPTION These constants are masks for different kinds of events. You pass them to NSCell's **sendActionOn:** method to indicate when an NSCell should send its action message.

NSCell—Data Entry Types

DECLARED IN AppKit/NSCell.h

SYNOPSIS

```
enum {
    NSAnyType,
    NSIntType,
    NSPositiveIntType,
    NSFloatType,
    NSPositiveFloatType,
    NSDoubleType,
    NSPositiveDoubleType
};
```

DESCRIPTION These constants represent the numeric data types that a text NSCell can accept. See NSCell's **setEntryType:** method for more information.

NSCell—States

DECLARED IN AppKit/NSCell.h

SYNOPSIS

```
enum {
    NSStateMixed,
    NSStateOff,
    NSStateOn
};
```

DESCRIPTION These constants are suggested parameter values for the NSCell method **setState:**.

NSColorPanel—Modes

DECLARED IN AppKit/NSColorPanel.h

SYNOPSIS

```
enum {
    NSGrayModeColorPanel,
    NSRGBModeColorPanel,
    NSCMYKModeColorPanel,
    NSHSBModeColorPanel,
    NSCustomPaletteModeColorPanel,
    NSColorListModeColorPanel,
    NSWheelModeColorPanel
};
```

DESCRIPTION These constants represent the possible modes of an NSColorPanel.

NSColorPanel—Mode Masks

DECLARED IN AppKit/NSColorPanel.h

SYNOPSIS

```
enum {
    NSColorPanelGrayModeMask,
    NSColorPanelRGBModeMask,
    NSColorPanelCMYKModeMask,
    NSColorPanelHSBModeMask,
    NSColorPanelCustomPaletteModeMask,
    NSColorPanelColorListModeMask,
    NSColorPanelWheelModeMask,
    NSColorPanelAllModesMask
};
```

DESCRIPTION These constants provide masks for the NSColorPanel modes.

NSDragging—Operations

DECLARED IN AppKit/NSDragging.h

SYNOPSIS

```
enum {
    NSDragOperationNone,
    NSDragOperationCopy,
    NSDragOperationLink,
    NSDragOperationGeneric,
    NSDragOperationPrivate,
    NSDragOperationAll
};
```

DESCRIPTION These constants define the operations that result from a user's drag. For full descriptions of their meanings and uses, see the method descriptions for **draggingSourceOperationMaskForLocal:** (in the NSDraggingSource protocol), **draggingSourceOperationMask** (in the NSDraggingInfo protocol), or **draggingEntered:** (in the NSDraggingDestination protocol).

NSEvent—Function-Key Unicodes

DECLARED IN AppKit/NSEvent.h

SYNOPSIS

```
enum {
    NSUpArrowFunctionKey = 0xF700,
    NSDownArrowFunctionKey = 0xF701,
    NSLeftArrowFunctionKey = 0xF702,
    NSRightArrowFunctionKey = 0xF703,
    NSF1FunctionKey = 0xF704,
    NSF2FunctionKey = 0xF705,
    NSF3FunctionKey = 0xF706,
    NSF4FunctionKey = 0xF707,
    NSF5FunctionKey = 0xF708,
    NSF6FunctionKey = 0xF709,
    NSF7FunctionKey = 0xF70A,
    NSF8FunctionKey = 0xF70B,
    NSF9FunctionKey = 0xF70C,
    NSF10FunctionKey = 0xF70D,
```

NSF11FunctionKey = 0xF70E,
NSF12FunctionKey = 0xF70F,
NSF13FunctionKey = 0xF710,
NSF14FunctionKey = 0xF711,
NSF15FunctionKey = 0xF712,
NSF16FunctionKey = 0xF713,
NSF17FunctionKey = 0xF714,
NSF18FunctionKey = 0xF715,
NSF19FunctionKey = 0xF716,
NSF20FunctionKey = 0xF717,
NSF21FunctionKey = 0xF718,
NSF22FunctionKey = 0xF719,
NSF23FunctionKey = 0xF71A,
NSF24FunctionKey = 0xF71B,
NSF25FunctionKey = 0xF71C,
NSF26FunctionKey = 0xF71D,
NSF27FunctionKey = 0xF71E,
NSF28FunctionKey = 0xF71F,
NSF29FunctionKey = 0xF720,
NSF30FunctionKey = 0xF721,
NSF31FunctionKey = 0xF722,
NSF32FunctionKey = 0xF723,
NSF33FunctionKey = 0xF724,
NSF34FunctionKey = 0xF725,
NSF35FunctionKey = 0xF726,
NSInsertFunctionKey = 0xF727,
NSDeleteFunctionKey = 0xF728,
NSHomeFunctionKey = 0xF729,
NSBeginFunctionKey = 0xF72A,
NSEndFunctionKey = 0xF72B,
NSPageUpFunctionKey = 0xF72C,
NSPageDownFunctionKey = 0xF72D,
NSPrintScreenFunctionKey = 0xF72E,
NSScrollLockFunctionKey = 0xF72F,
NSPauseFunctionKey = 0xF730,
NSSysReqFunctionKey = 0xF731,
NSBreakFunctionKey = 0xF732,
NSResetFunctionKey = 0xF733,
NSStopFunctionKey = 0xF734,
NSMenuFunctionKey = 0xF735,
NSUserFunctionKey = 0xF736,
NSSystemFunctionKey = 0xF737,
NSPrintFunctionKey = 0xF738,

NSClearLineFunctionKey = 0xF739,
NSClearDisplayFunctionKey = 0xF73A,
NSInsertLineFunctionKey = 0xF73B,
NSDeleteLineFunctionKey = 0xF73C,
NSInsertCharFunctionKey = 0xF73D,
NSDeleteCharFunctionKey = 0xF73E,
NSPrevFunctionKey = 0xF73F,
NSNextFunctionKey = 0xF740,
NSSelectFunctionKey = 0xF741,
NSExecuteFunctionKey = 0xF742,
NSUndoFunctionKey = 0xF743,
NSRedoFunctionKey = 0xF744,
NSFindFunctionKey = 0xF745,
NSHelpFunctionKey = 0xF746,
NSModeSwitchFunctionKey = 0xF747
};

DESCRIPTION These Unicodes (0xF700-0xF8FF) are reserved for function keys on the keyboard. Combined in NSStrings, they may be used in the return value of the NSEvent methods **characters...**, and in parameters of the NSEvent method **keyEventWithType:...characters:...**.

NSEvent—Modifier Flags

DECLARED IN AppKit/NSEvent.h

SYNOPSIS

enum {
NSAlphaShiftKeyMask,
NSShiftKeyMask,
NSControlKeyMask,
NSAlternateKeyMask,
NSCommandKeyMask,
NSNumericPadKeyMask,
NSHelpKeyMask,
NSFunctionKeyMask
};

DESCRIPTION These are device-independent bits found in event modifier flags.

NSEvent—Types Defined by the Application Kit

DECLARED IN AppKit/NSEvent.h

SYNOPSIS

```
enum {
    NSWindowExposedEventType,
    NSApplicationActivatedEventType,
    NSApplicationDeactivatedEventType,
    NSWindowMovedEventType,
    NSScreenChangedEventType
};
```

DESCRIPTION These constants represent the types of events defined by the Application Kit.

Constant	Meaning
NSWindowExposedEventType	A nonretained NSWindow has been exposed
NSApplicationActivatedEventType	The application has been activated
NSApplicationDeactivatedEventType	The application has been deactivated
NSWindowMovedEventType	An NSWindow has moved
NSScreenChangedEventType	An NSWindow has changed screens

NSEvent—Types Defined by the System

DECLARED IN AppKit/NSEvent.h

SYNOPSIS

```
enum {
    NSPowerOffEventType
};
```

DESCRIPTION This constant means that the user is turning off the computer.

NSFont—Traits

DECLARED IN AppKit/NSFontManager.h

SYNOPSIS

```
enum {
    NSItalicFontMask,
    NSBoldFontMask,
    NSUnboldFontMask,
    NSNonStandardCharacterSetFontMask,
    NSNarrowFontMask,
    NSExpandedFontMask,
    NSCondensedFontMask,
    NSSmallCapsFontMask,
    NSPosterFontMask,
    NSCompressedFontMask,
    NSFixedPitchFontMask,
    NSUnitalicFontMask
};
```

DESCRIPTION These constants are used by the NSFontManager to identify font traits. Some traits are mutually exclusive, such as NSExpandedFontMask and NSCondensedFontMask.

NSFontPanel—Tags for Subviews

DECLARED IN AppKit/NSFontPanel.h

SYNOPSIS

```
enum {
    NSFPPreviewButton,
    NSFPRevertButton,
    NSFPSetButton,
    NSFPPreviewField,
    NSFPSizeField,
    NSFPSizeTitle,
    NSFPCurrentField
};
```

DESCRIPTION These tags identify the NSViews within an NSFontPanel.

NSGlyph—Attributes

DECLARED IN AppKit/NSLayoutManager.h

SYNOPSIS

```
enum _NSGlyphAttribute {
    NSGlyphAttributeSoft,
    NSGlyphAttributeElastic,
    NSGlyphAttributeInscribe
};
```

DESCRIPTION These glyph attributes are used only inside the glyph generation machinery, but must be shared between components.

NSGlyph—Reserved Glyph Codes

DECLARED IN AppKit/NSFont.h

SYNOPSIS

```
enum {
    NSControlGlyph = 0x00FFFFFF,
    NSNullGlyph = 0x0
};
```

DESCRIPTION These two values are reserved for the two named NSGlyphs.

NSImageRep—Display Device Matching

DECLARED IN AppKit/NSImageRep.h

SYNOPSIS

```
enum {
    NSImageRepMatchesDevice
};
```

DESCRIPTION This constant is used by NSImageRep to indicate that the value of certain attributes, such as the number of colors, or bits-per-sample, will change to match the display device. See the NSImageRep class specification for more information.

NSPageLayoutPanel—Tags for Controls

DECLARED IN AppKit/NSPageLayout.h

SYNOPSIS

```
enum {
    NSPLImageButton,
    NSPLTitleField,
    NSPLPaperNameButton,
    NSPLUnitsButton,
    NSPLWidthForm,
    NSPLHeightForm,
    NSPLOrientationMatrix,
    NSPLCancelButton,
    NSPLOKButton
};
```

DESCRIPTION These constants represent the tag values of the controls displayed by an NSPageLayoutPanel.

NSPanel—Alert Panel Return Values

DECLARED IN AppKit/NSPanel.h

SYNOPSIS

```
enum {
    NSAlertDefaultReturn,
    NSAlertAlternateReturn,
    NSAlertOtherReturn,
    NSAlertErrorReturn
};
```

DESCRIPTION These constants define values returned by the **NSRunAlertPanel** function and by the NSApplication method **runModalSession:** when the modal session is run with an NSPanel provided by the **NSGetAlertPanel** function.

NSPanel—Modal Panel Return Values

DECLARED IN AppKit/NSPanel.h

SYNOPSIS

```
enum {
    NSOKButton,
    NSCancelButton
};
```

DESCRIPTION These are the possible return values for such methods as the **runModal...** methods of NSOpenPanel, which tells which button (OK or Cancel) the user has clicked on an open panel. For other uses of these return values, see the class descriptions for NSPageLayout, NSPrintPanel and NSSavePanel.

NSPrintPanel—Tags for Subviews

DECLARED IN AppKit/NSPrintPanel.h

SYNOPSIS

```
enum {
    NSPPSaveButton,
    NSPPPreviewButton,
    NSFaxButton,
    NSPPTitleField,
    NSPPImageButton,
    NSPPNameTitle,
    NSPPNameField,
    NSPPNoteTitle,
    NSPPNoteField,
    NSPPStatusTitle,
    NSPPStatusField,
    NSPPCopiesField,
    NSPPPageChoiceMatrix,
    NSPPPageRangeFrom,
    NSPPPageRangeTo,
    NSPPScaleField,
    NSPPOptionsButton,
    NSPPPaperFeedButton,
    NSPPLayoutButton
};
```

DESCRIPTION These constants define tags for identifying the NSViews in a print panel in environments other than Microsoft Windows. Windows has its own way of handling print panels.

NSRunLoop—Ordering Modes for NSApplication

DECLARED IN AppKit/NSApplication.h

SYNOPSIS

```
enum {
    NSUpdateWindowsRunLoopOrdering
};
```

DESCRIPTION This constants is used with NSRunLoop's **performSelector:target:argument:order:modes:** method.

NSRunLoop—Ordering Mode for NSDPSServerContext

DECLARED IN AppKit/NSDPSServerContext.h

SYNOPSIS

```
enum {
    DPSFlushContextRunLoopOrdering
};
```

DESCRIPTION This constants is used with NSRunLoop's method **performSelector:target:argument:order:modes:**.

NSRunLoop—Ordering Modes for NSWindow

DECLARED IN AppKit/NSWindow.h

SYNOPSIS

```
enum {
    NSDisplayWindowRunLoopOrdering,
    NSResetCursorRectsRunLoopOrdering
};
```

DESCRIPTION These constants are passed to NSRunLoop’s method **performSelector:target:argument:order:modes:**.

NSSavePanel—Tags for Subviews

DECLARED IN AppKit/NSSavePanel.h

SYNOPSIS

```
enum {
    NSFileHandlingPanelImageButton,
    NSFileHandlingPanelTitleField,
    NSFileHandlingPanelBrowser,
    NSFileHandlingPanelCancelButton,
    NSFileHandlingPanelOKButton,
    NSFileHandlingPanelForm,
    NSFileHandlingPanelHomeButton,
    NSFileHandlingPanelDiskButton,
    NSFileHandlingPanelDiskEjectButton
};
```

DESCRIPTION These constants define tags for identifying NSViews in an NSSavePanel.

NSTextAttachment—Attachment Character

DECLARED IN AppKit/NSTextAttachment.h

SYNOPSIS

```
enum {
    NSAttachmentCharacter = 0xfffc
};
```

DECLARED IN This Unicode indicates the presence of an attachment in an NSAttributedString. For more information, see the Class Cluster Description of NSAttributedStringAdditions.

NSText—Important Unicodes

DECLARED IN AppKit/NSText.h

SYNOPSIS

```
enum {
    NSParagraphSeparatorCharacter = 0x2029,
    NSLineSeparatorCharacter = 0x2028,
    NSTabCharacter = 0x0009,
    NSFormFeedCharacter = 0x000c,
    NSNewlineCharacter = 0x000a,
    NSCarriageReturnCharacter = 0x000d,
    NSEnterCharacter = 0x0003,
    NSBackspaceCharacter = 0x0008,
    NSBackTabCharacter = 0x0019,
    NSDeleteCharacter = 0x007f,
};
```

DESCRIPTION These constants specify several commonly-used Unicode characters.

NSText—Movement Codes

DECLARED IN AppKit/NSText.h

SYNOPSIS

```
enum {
    NSIllegalTextMovement,
    NSReturnTextMovement,
    NSTabTextMovement,
    NSBacktabTextMovement,
    NSLeftTextMovement,
    NSRightTextMovement,
    NSUpTextMovement,
    NSDownTextMovement
};
```

DESCRIPTION These constants are the codes for movement between fields. They are the possible int values for the NSTextMovement key of NSTextDidEndEditingNotification. For more information, see the "Notifications" section of the NSText class specification.

NSTextStorage—Editing

DECLARED IN AppKit/NSTextStorage.h

SYNOPSIS

```
enum {
    NSTextStorageEditedAttributes,
    NSTextStorageEditedCharacters
};
```

DESCRIPTION These values, which may be combined by a bitwise OR, help describe the changes that an editing session has made to an NSTextStorage object. They are the return values of the NSTextStorage method **editedMask**, and the parameter values for the second slot of the NSLayoutManager method **textStorage:edited:...**.

NSView—Resizing

DECLARED IN AppKit/NSView.h

SYNOPSIS

```
enum {
    NSViewNotSizable,
    NSViewMinXMargin,
    NSViewWidthSizable,
    NSViewMaxXMargin,
    NSViewMinYMargin,
    NSViewHeightSizable,
    NSViewMaxYMargin
};
```

DESCRIPTION Used to describe which parts of an NSView (or its margins) are resized when the NSView's superNSView is resized. See the NSView class specification for details.

NSWindow—Border Masks

DECLARED IN AppKit/NSWindow.h

SYNOPSIS

```
enum {
    NSBorderlessWindowMask,
    NSTitledWindowMask,
    NSClosableWindowMask,
    NSMiniaturizableWindowMask,
    NSResizableWindowMask
};
```

DESCRIPTION These determine the presence of a title and various buttons in an NSWindow's border.

NSWindow—Window Levels

DECLARED IN AppKit/NSWindow.h

SYNOPSIS

```
enum {
    NSNormalWindowLevel,
    NSFloatingWindowLevel,
    NSDockWindowLevel,
    NSSubmenuWindowLevel,
    NSTornOffMenuWindowLevel,
    NSMainMenuWindowLevel,
    NSModalPanelWindowLevel,
    NSPopUpMenuWindowLevel
};
```

DESCRIPTION These constants name the Application Kit's window levels. The stacking of levels takes precedence over the stacking of windows within each level. That is, even the bottom window in a level will obscure even the top window of the next level down.

The constant NSTornOffMenuWindowLevel is preferable to its synonym, NSSubmenuWindowLevel.

Global Variables

Application Kit—Exceptions

DECLARED IN AppKit/NSErrors.h

SYNOPSIS

NSString ***NSTextLineTooLongException**;
NSString ***NSTextNoSelectionException**;
NSString ***NSWordTablesWriteException**;
NSString ***NSWordTablesReadException**;
NSString ***NSTextReadException**;
NSString ***NSTextWriteException**;
NSString ***NSPasteboardCommunicationException**;
NSString ***NSPrintingCommunicationException**;
NSString ***NSAbortModalException**;
NSString ***NSAbortPrintingException**;
NSString ***NSIllegalSelectorException**;
NSString ***NSAppKitVirtualMemoryException**;
NSString ***NSBadRTFDirectiveException**;
NSString ***NSBadRTFFontTableException**;
NSString ***NSBadRTFStyleSheetException**;
NSString ***NSTypedStreamVersionException**;
NSString ***NSTIFFException**;
NSString ***NSPrintPackageException**;
NSString ***NSBadRTFColorTableException**;
NSString ***NSDraggingException**;

NSString ***NSColorListIOException**;
NSString ***NSColorListNotEditableException**;
NSString ***NSBadBitmapParametersException**;
NSString ***NSWindowServerCommunicationException**;
NSString ***NSFontUnavailableException**;
NSString ***NSPPDIncludeNotFoundException**;
NSString ***NSPPDParseException**;
NSString ***NSPPDIncludeStackOverflowException**;
NSString ***NSPPDIncludeStackUnderflowException**;
NSString ***NSRTFPropertyStackOverflowException**;
NSString ***NSAppKitIgnoredException**;
NSString ***NSBadComparisonException**;
NSString ***NSImageCacheException**;
NSString ***NSNibLoadingException**;
NSString ***NSBrowserIllegalDelegateException**;

DESCRIPTION These constants name the exceptions that the Application Kit can raise.

Display Device—Descriptions

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS
NSString ***NSDeviceResolution**;
NSString ***NSDeviceColorSpaceName**;
NSString ***NSDeviceBitsPerSample**;
NSString ***NSDeviceIsScreen**;
NSString ***NSDeviceIsPrinter**;
NSString ***NSDeviceSize**;

DESCRIPTION These are the keys for device description dictionaries, such as those returned by the **deviceDictionary** methods of NSPrinter, NSScreen and NSWindow.

NSDeviceResolution is an NSValue containing an NSSize in dots per inch. NSColorSpaceName is an NSString describing the color space of the device. NSDeviceBitsPerSample is an NSValue containing an int. NSDeviceIsScreen and NSDeviceIsPrinter are boolean values that tell whether the device is a screen or a printer. NSDeviceSize is an NSValue containing an NSSize that represents the device's size in points.

NSApplication—Notifications

DECLARED IN AppKit/NSApplication.h

SYNOPSIS

NSString ***NSApplicationDidBecomeActiveNotification**;
NSString ***NSApplicationDidFinishLaunchingNotification**;
NSString ***NSApplicationDidHideNotification**;
NSString ***NSApplicationDidResignActiveNotification**;
NSString ***NSApplicationDidUnhideNotification**;
NSString ***NSApplicationDidUpdateNotification**;
NSString ***NSApplicationWillBecomeActiveNotification**;
NSString ***NSApplicationWillFinishLaunchingNotification**;
NSString ***NSApplicationWillHideNotification**;
NSString ***NSApplicationWillResignActiveNotification**;
NSString ***NSApplicationWillUnhideNotification**;
NSString ***NSApplicationWillUpdateNotification**;
NSString ***NSApplicationWillTerminateNotification**;

DECLARED IN These are the notifications used with the methods of the NSApplicationNotifications category of NSObject.

NSApplication—Shared Application Object

DECLARED IN AppKit/NSApplication.h

SYNOPSIS id **NSApp**;

DESCRIPTION This variable designates the shared application object, created by NSApplication's **sharedApplication** method.

NSAttributedString—Attributes

DECLARED IN AppKit/NSAttributedString.h

SYNOPSIS

NSString ***NSFontAttributeName**;
NSString ***NSParagraphStyleAttributeName**;
NSString ***NSForegroundColorAttributeName**;
NSString ***NSUnderlineStyleAttributeName**;
NSString ***NSSuperscriptAttributeName**;
NSString ***NSBackgroundColorAttributeName**;
NSString ***NSAttachmentAttributeName**;
NSString ***NSLigatureAttributeName**;
NSString ***NSBaselineOffsetAttributeName**;
NSString ***NSKernAttributeName**;

DESCRIPTION These strings define the supported attributes of NSAttributedStrings. For more information, see the “Accessing Attributes” section in the NSAttributedString class cluster specification.

NSComboBox—Notifications

DECLARED IN AppKit/NSComboBox.h

SYNOPSIS

NSString ***NSComboBoxWillPopUpNotification**;
NSString ***NSComboBoxWillDismissNotification**;
NSString ***NSComboBoxSelectionDidChangeNotification**;
NSString ***NSComboBoxSelectionIsChangingNotification**;

DESCRIPTION These notifications are sent by NSComboBoxes.

NSColor—Color Space Names

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS

NSString ***NSCalibratedWhiteColorSpace**;
NSString ***NSCalibratedBlackColorSpace**;
NSString ***NSCalibratedRGBColorSpace**;
NSString ***NSDeviceWhiteColorSpace**;
NSString ***NSDeviceBlackColorSpace**;
NSString ***NSDeviceRGBColorSpace**;
NSString ***NSDeviceCMYKColorSpace**;
NSString ***NSNamedColorSpace**;
NSString ***NSCustomColorSpace**;

DESCRIPTION These are the predefined names for color spaces. In the two ...WhiteColorSpaces, white corresponds to a value of 1.0. In the two ...BlackColorSpaces, black corresponds to a value of 1.0. NSNamedColorSpace is used for “catalog” colors—that is, colors specified by names rather than coordinates. NSCustomColorSpace indicates a custom color space, which can be useful in working with images; unlike the other color spaces, NSCustomColorSpace is not used with NSColors.

NSColor—Grayscale Values

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS

const float **NSWhite**;
const float **NSLightGray**;
const float **NSDarkGray**;
const float **NSBlack**;

DESCRIPTION These are the standard gray values for the 2-bit deep grayscale color space.

NSColor—Notifications

DECLARED IN AppKit/NSColor.h

SYNOPSIS NSString ***NSSystemColorsDidChangeNotification**;

DESCRIPTION This notification is sent when the system colors have been changed (such as through a system control panel interface). For more on system colors, see the “System Colors” section of the NSColor class specification.

NSColorList—Notifications

DECLARED IN AppKit/NSColorList.h

SYNOPSIS NSString ***NSColorListDidChangeNotification**;

DESCRIPTION When an NSColorList changes, it posts this notification.

NSColorPanel—Notifications

DECLARED IN AppKit/NSColorPanel.h

SYNOPSIS NSString ***NSColorPanelColorDidChangeNotification**;

DESCRIPTION When an NSColorPanel changes, it posts this notification.

NSControl—Notifications

DECLARED IN AppKit/NSControl.h

SYNOPSIS NSString ***NSControlTextDidBeginEditingNotification**;
NSString ***NSControlTextDidEndEditingNotification**;
NSString ***NSControlTextDidChangeNotification**;

DESCRIPTION NSControls containing editable text can send these notifications. For more information, see the "Notifications" section of the NSControl class specification.

NSDataLink—Filename Extension

DECLARED IN AppKit/NSDataLink.h

SYNOPSIS NSString ***NSDataLinkFilenameExtension**;

DESCRIPTION NSDataLinkFilenameExtension is the filename extension used for links saved to files using NSDataLink's **saveLinkIn:** or **writeToFile:** methods.

NSFont—Keys to the AFM Dictionary

DECLARED IN AppKit/NSFont.h

SYNOPSIS

NSString ***NSAFMFamilyName**;
NSString ***NSAFMFontName**;
NSString ***NSAFMFormatVersion**;
NSString ***NSAFMFullName**;
NSString ***NSAFMNotice**;
NSString ***NSAFMVersion**;
NSString ***NSAFMWeight**;
NSString ***NSAFMEncodingScheme**;
NSString ***NSAFMCharacterSet**;
NSString ***NSAFMCapHeight**;
NSString ***NSAFMXHeight**;
NSString ***NSAFMAscender**;
NSString ***NSAFMDescender**;
NSString ***NSAFMUnderlinePosition**;
NSString ***NSAFMUnderlineThickness**;
NSString ***NSAFMItalicAngle**;
NSString ***NSAFMMappingScheme**;

DESCRIPTION These are the keys to the font information dictionary returned by NSFont's **afmDictionary** method. To convert values like NSAFMCapHeight to floats, use NSString's **floatValue** method.

For other font information, use NSFont's **afmFileContents** method.

NSFont—PostScript Transformation Matrix

DECLARED IN AppKit/NSFont.h

SYNOPSIS

const float ***NSFontIdentityMatrix**;

DESCRIPTION NSFontIdentityMatrix is a PostScript transformation matrix useful as a parameter to the NSFont method **fontWithName:matrix:**.

NSHelpManager—Notifications

DECLARED IN AppKit/NSHelpManager.h

SYNOPSIS NSString ***NSContextHelpModeDidActivateNotification**;
NSString ***NSContextHelpModeDidDeactivateNotification**;

DESCRIPTION These are notifications for the activation and deactivation of the context help mode.

NSImageRep—Notifications

DECLARED IN AppKit/NSImageRep.h

SYNOPSIS NSString ***NSImageRepRegistryDidChangeNotification**;

DESCRIPTION This notification is sent when the NSImageRep class registry changes.

NSInterfaceStyleDefault

DECLARED IN AppKit/NSInterfaceStyle.h

SYNOPSIS NSString ***NSInterfaceStyleDefault**;

DESCRIPTION NSInterfaceStyleDefault can be used to override the platform's default interface style. For more information, see the function **NSInterfaceStyleForKey**.

NSPasteboard—Names

DECLARED IN AppKit/NSPasteboard.h

SYNOPSIS
NSString ***NSGeneralPboard**;
NSString ***NSFontPboard**;
NSString ***NSRulerPboard**;
NSString ***NSFindPboard**;
NSString ***NSDragPboard**;

DESCRIPTION Some standard pasteboard names. See the NSPasteboard class specification for more information.

NSPasteboard—Type for Data Links

DECLARED IN AppKit/NSDataLink.h

SYNOPSIS NSString ***NSDataLinkPboardType**;

DESCRIPTION A pasteboard type for copying a data link to the pasteboard. See the NSDataLink class specification for more information.

NSPasteboard—Type for Selection Descriptions

DECLARED IN AppKit/NSSelection.h

SYNOPSIS NSString ***NSSelectionPboardType**;

DESCRIPTION A pasteboard type for copying selection descriptions to the pasteboard. See the NSSelection class specification for more information.

NSPasteboard—Types for Standard Data

DECLARED IN AppKit/NSPasteboard.h

SYNOPSIS

NSString ***NSStringPboardType**;
NSString ***NSFilenamesPboardType**;
NSString ***NSPostScriptPboardType**;
NSString ***NSTIFFPboardType**;
NSString ***NSRTFPboardType**;
NSString ***NSTabularTextPboardType**;
NSString ***NSFontPboardType**;
NSString ***NSRulerPboardType**;
NSString ***NSFileContentsPboardType**;
NSString ***NSColorPboardType**;
NSString ***NSRTFDPboardType**;

DESCRIPTION Some standard pasteboard data types. See the NSPasteboard class specification for more information.

NSPrintInfo—Dictionary Keys

DECLARED IN AppKit/NSPrintInfo.h

SYNOPSIS

NSString ***NSPrintPaperName**;
NSString ***NSPrintPaperSize**;
NSString ***NSPrintFormName**;
NSString ***NSPrintMustCollate**;
NSString ***NSPrintOrientation**;
NSString ***NSPrintLeftMargin**;
NSString ***NSPrintRightMargin**;
NSString ***NSPrintTopMargin**;
NSString ***NSPrintBottomMargin**;
NSString ***NSPrintHorizontallyCentered**;
NSString ***NSPrintVerticallyCentered**;
NSString ***NSPrintHorizontalPagination**;
NSString ***NSPrintVerticalPagination**;
NSString ***NSPrintScalingFactor**;
NSString ***NSPrintAllPages**;
NSString ***NSPrintReversePageOrder**;
NSString ***NSPrintFirstPage**;
NSString ***NSPrintLastPage**;
NSString ***NSPrintCopies**;
NSString ***NSPrintPagesPerSheet**;
NSString ***NSPrintJobFeatures**;
NSString ***NSPrintPaperFeed**;
NSString ***NSPrintManualFeed**;
NSString ***NSPrintPrinter**;
NSString ***NSPrintJobDisposition**;

NSString ***NSPrintSavePath**;
NSString ***NSPrintFaxReceiverNames**;
NSString ***NSPrintFaxReceiverNumbers**;
NSString ***NSPrintFaxSendTime**;
NSString ***NSPrintFaxUseCoverSheet**;
NSString ***NSPrintFaxCoverSheetName**;
NSString ***NSPrintFaxReturnReceipt**;
NSString ***NSPrintFaxHighResolution**;
NSString ***NSPrintFaxTrimPageEnds**;
NSString ***NSPrintFaxModem**;
NSString ***NSPrintSpoolJob**;
NSString ***NSPrintFaxJob**;
NSString ***NSPrintPreviewJob**;
NSString ***NSPrintSaveJob**;
NSString ***NSPrintCancelJob**;

DESCRIPTION These are the keys to the NSPrintInfo NSDictionary. For a table explaining them, see the NSPrintInfo method **initWithDictionary:**.

NSPopUpButton—Notification

DECLARED IN AppKit/NSPopUpButton.h

SYNOPSIS NSString ***NSPopUpButtonWillPopUpNotification**;

DESCRIPTION NSPopUpButton sends this notification when an instance of it is about to pop up.

NSPrintOperation—Exception

DECLARED IN AppKit/NSPrintOperation.h

SYNOPSIS NSString ***NSPrintOperationExistsException**;

DESCRIPTION This exception is raised when there is already a print operation in process. The methods that raise it are the **EPSOperation...** and **printOperation...** methods in NSPrintOperation:

NSRunLoop—Modes

DECLARED IN AppKit/NSApplication.h

SYNOPSIS NSString ***NSModalPanelRunLoopMode**;
NSString ***NSEventTrackingRunLoopMode**;

DESCRIPTION These are modes passed to NSRunLoop

NSSplitView—Notifications

DECLARED IN AppKit/NSSplitView.h

SYNOPSIS NSString ***NSSplitViewDidResizeSubviewsNotification**;
NSString ***NSSplitViewWillResizeSubviewsNotification**;

DESCRIPTION These are the notifications that an NSSplitView can send.

NSTableView—Notifications

DECLARED IN AppKit/NSTableView.h

SYNOPSIS NSString ***NSTableViewSelectionDidChangeNotification**;
NSString ***NSTableViewColumnDidMoveNotification**;
NSString ***NSTableViewColumnDidResizeNotification**;
NSString ***NSTableViewSelectionIsChangingNotification**;

DESCRIPTION These are the notifications that an NSTableView can send.

NSText—Notifications

DECLARED IN AppKit/NSText.h

SYNOPSIS NSString ***NSTextDidBeginEditingNotification**;
NSString ***NSTextDidEndEditingNotification**;
NSString ***NSTextDidChangeNotification**;

DESCRIPTION These notifications can be sent by an NSText object. For explanations, see the “Notifications” section of the NSText class specification.

NSTextStorage—Notifications

DECLARED IN AppKit/NSTextStorage.h

SYNOPSIS NSString ***NSTextStorageWillProcessEditingNotification**;
NSString ***NSTextStorageDidProcessEditingNotification**;

DESCRIPTION These notifications can be sent by an NSTextStorage object. For explanations, see the “Notifications” section of the NSTextStorage class specification.

NSTextView—Notifications

DECLARED IN AppKit/NSTextView.h

SYNOPSIS NSString ***NSTextViewWillChangeNotifyingTextViewNotification**;
NSString ***NSTextViewDidChangeSelectionNotification**;

DESCRIPTION These notifications can be sent by an NSTextView object. For explanations, see the “Notifications” section of the NSTextView class specification.

The notifications that NSTextView most often sends are the ones that it inherits from NSText.

NSView—Notifications

DECLARED IN AppKit/NSView.h

SYNOPSIS
NSString ***NSViewFrameDidChangeNotification**;
NSString ***NSViewFocusDidChangeNotification**;
NSString ***NSViewBoundsDidChangeNotification**;

DESCRIPTION These notifications are sent by NSViews.

The last notification, NSViewBoundsDidChangeNotification, is sent when the view bounds change but the frame does not. That is, it is sent whenever the view's bounds are translated, scaled or rotated, but not when the bounds change in response to, say, a **setFrameSize:** message.

NSWindow—Notifications

DECLARED IN AppKit/NSWindow.h

SYNOPSIS
NSString ***NSWindowDidBecomeKeyNotification**;
NSString ***NSWindowDidBecomeMainNotification**;
NSString ***NSWindowDidChangeScreenNotification**;
NSString ***NSWindowDidDeminiaturizeNotification**;
NSString ***NSWindowDidExposeNotification**;
NSString ***NSWindowDidMiniaturizeNotification**;
NSString ***NSWindowDidMoveNotification**;
NSString ***NSWindowDidResignKeyNotification**;
NSString ***NSWindowDidResignMainNotification**;
NSString ***NSWindowDidResizeNotification**;
NSString ***NSWindowDidUpdateNotification**;
NSString ***NSWindowWillCloseNotification**;
NSString ***NSWindowWillMiniaturizeNotification**;
NSString ***NSWindowWillMoveNotification**;

DESCRIPTION These are the notifications that can be sent by an NSWindow object. For explanations, see the “Notifications” section of the NSWindow class specification.

NSWindow—Sizes

DECLARED IN AppKit/NSWindow.h

SYNOPSIS

NSSize **NSIconSize**;
NSSize **NSTokenSize**;

DESCRIPTION On some platforms, a token is a bezeled tile used to represent a docked application or a miniaturized document, and an icon is the image drawn inside a token.

On platforms that support tokens and icons, these size constants can be used for drawing inside them. It is more portable, however, to change an icon by using the NSApplication method **setApplicationIconImage:** or the NSWindow method **setMiniwindowImage:**.

NSWorkspace—File Operation Constants

DECLARED IN AppKit/NSWorkspace.h

SYNOPSIS

NSString ***NSWorkspaceMoveOperation**;
NSString ***NSWorkspaceCopyOperation**;
NSString ***NSWorkspaceLinkOperation**;
NSString ***NSWorkspaceCompressOperation**;
NSString ***NSWorkspaceDecompressOperation**;
NSString ***NSWorkspaceEncryptOperation**;
NSString ***NSWorkspaceDecryptOperation**;
NSString ***NSWorkspaceDestroyOperation**;
NSString ***NSWorkspaceRecycleOperation**;
NSString ***NSWorkspaceDuplicateOperation**;

DESCRIPTION These constants define possible values for the *operation* slot in NSWorkspace's **performFileOperation:**... method.

NSWorkspace—File Types

DECLARED IN AppKit/NSWorkspace.h

SYNOPSIS
NSString ***NSPlainFileType**;
NSString ***NSDirectoryFileType**;
NSString ***NSApplicationFileType**;
NSString ***NSFilesystemFileType**;
NSString ***NSShellCommandFileType**;

DESCRIPTION These values are used in the final parameter slot of the NSWorkspace method **getInfoForFile: application:type:**.

NSWorkspace—Notifications

DECLARED IN AppKit/NSWorkspace.h

SYNOPSIS
NSString ***NSWorkspaceDidLaunchApplicationNotification**;
NSString ***NSWorkspaceDidMountNotification**;
NSString ***NSWorkspaceDidPerformFileOperationNotification**;
NSString ***NSWorkspaceDidTerminateApplicationNotification**;
NSString ***NSWorkspaceDidUnmountNotification**;
NSString ***NSWorkspaceWillLaunchApplicationNotification**;
NSString ***NSWorkspaceWillPowerOffNotification**;
NSString ***NSWorkspaceWillUnmountNotification**;

DESCRIPTION These notifications come through the special notification center. For more information, see the "Notifications" section of the NSWorkspace class specification.

Defined Types

Note: This section has not been updated and has not received recent technical review. It is included in this release to test the linkage between application development tools and on-line documentation. The information in this section should be considered at best preliminary and subject to change. An updated version of this file will be included in the next release.

DPSContextRec

DECLARED IN dpsclient/dpsfriends.h

SYNOPSIS

```
typedef struct _t_DPSContextRec {
        char *priv;
        DPSSpace space;
        DPSProgramEncoding programEncoding;
        DPSNameEncoding nameEncoding;
        struct _t_DPSProcsRec const * procs;
        void (*textProc)();
        void (*errorProc)();
        DPSResults resultTable;
        unsigned int resultTableLength;
        struct _t_DPSContextRec *chainParent, *chainChild;
        DPSContextType type;
} DPSContextRec, *DPSContext;
```

DESCRIPTION The **DPSContextRec** structure represents a Display PostScript context.

DPSContextType

DECLARED IN dpsclient/dpsfriends.h

SYNOPSIS

typedef enum {
    dps_machServer,
    dps_fdServer,
    dps_stream
} **DPSContextType**;

DESCRIPTION These represent the context types supported by NeXT's version of Display PostScript, as used in the **type** field of a **DPSContextRec** structure.

DPSErrorCode

DECLARED IN dpsclient/dpsclient.h

SYNOPSIS

typedef enum _DPSErrorCode {
    dps_err_ps = DPS_ERROR_BASE,
    dps_err_nameTooLong,
    dps_err_resultTagCheck,
    dps_err_resultTypeCheck,
    dps_err_invalidContext,
    dps_err_select = DPS_NEXT_ERROR_BASE,
    dps_err_connectionClosed,
    dps_err_read,
    dps_err_write,
    dps_err_invalidFD,
    dps_err_invalidTE,
    dps_err_invalidPort,
    dps_err_outOfMemory,
    dps_err_cantConnect
} **DPSErrorCode**;

DESCRIPTION Error codes passed to a **DPSErrorProc()** function.

DPSEventFilterFunc

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS typedef int (*__DPSEventFilterFunc__)(NXEvent **ev*);

DESCRIPTION Call-back function used to filter events.

DPSFDProc

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS typedef void (*__DPSFDProc__)(int *fd*, void **userData*);

DESCRIPTION Call-back function used when a file descriptor is registered through **DPSAddFD()**.

DPSNumberFormat

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS

```
typedef enum _DPSNumberFormat {
#ifdef __BIG_ENDIAN__
        dps_float = 48,
        dps_long = 0,
        dps_short = 32
#else
        dps_float = 48+128,
        dps_long = 0+128,
        dps_short = 32+128
} DPSNumberFormat;
```

DESCRIPTION These constants are used by the **DPSDoUserPath()** function to describe the type of numbers that are being passed.

DPSPingProc

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS

typedef void (*__DPSPingProc__)
(DPSContext *ctxt*,
void **userData*);

DESCRIPTION Call-back function used by **DPSAsynchronousWaitContext()**.

DPSPortProc

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS

typedef void (*__DPSPortProc__)
(msg_header_t **msg*,
void **userData*);

DESCRIPTION Call-back function used when a port is registered through **DPSAddPort()**.

DPSTimedEntry

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS

typedef struct __DPSTimedEntry ***DPSTimedEntry**;

DESCRIPTION The return type for **DPSAddTimedEntry()**.

DPSTimedEntryProc

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS

typedef void (*__DPSTimedEntryProc__)
(DPSTimedEntry *timedEntry*,
double *now*,
void **userData*);

DESCRIPTION Call-back function used when a timed entry is registered through **DPSAddTimedEntry()**.

DPSUserPathAction

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS

typedef enum _DPSUserPathAction {
dps_uappend,
dps_ufill,
dps_ueofill,
dps_ustroke,
dps_ustrokepath,
dps_inufill,
dps_inueofill,
dps_inustroke,
dps_def,
dps_put
} **DPSUserPathAction**;

DESCRIPTION These constants are convenient representations of some of the PostScript operator indices, suitable for enrollment in the action array passed to **DPSDoUserPath()**.

DPSUserPathOp

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS

```
typedef enum _DPSUserPathOp {
        dps_setbbox,
        dps_moveto,
        dps_rmoveto,
        dps_lineto,
        dps_rlineto,
        dps_curveto,
        dps_rcurveto,
        dps_arc,
        dps_arcn,
        dps_arct,
        dps_closepath,
        dps_ucache
} DPSUserPathOp;
```

DESCRIPTION These constants represent the PostScript operators that can be passed in **DPSDoUserPath()**'s operator array.

NXCoord

DECLARED IN dpsclient/event.h

SYNOPSIS typedef float **NXCoord**

DESCRIPTION Used to represent a single coordinate in a Cartesian coordinate system.

NXEvent

DECLARED IN dpsclient/event.h

SYNOPSIS

```
typedef struct _NXEvent {
        int type;
        NXPoint location;
        long time;
        int flags;
        unsigned int window;
        NXEventData data;
        DPSContext ctxt;
} NXEvent, *NXEventPtr;
```

DESCRIPTION Represents a single event; this structure is also known as the *event record*. The fields are:

type	The type of event (see “Event Types,” below)
location	The event’s location in the base coordinate system of its window
time	The time of the event (in hardware-dependent units) since system startup
flags	Mouse-button and modifier-key flags (see “Event Flags,” below)
window	The window number of the window associated with the event
data	Additional type-specific data (see “NXEventData,” below)
ctxt	The PostScript context of the event

NXEventData

DECLARED IN dpsclient/event.h

SYNOPSIS

```
typedef union {
    struct {
        short eventNum;
        int click;
        unsigned char pressure;
    } mouse;
    struct {
        short repeat;
        unsigned short charSet;
        unsigned short charCode;
        unsigned short keyCode;
        short keyData;
    } key;
    struct {
        short eventNum;
        int trackingNum;
        int userData;
    } tracking;
    struct {
        short subtype;
        union {
            float F[2];
            long L[2];
            short S[4];
            char C[8];
        } misc;
    } compound;
} NXEventData;
```

DESCRIPTION This structure supplies type-specific information for an event. It's a union of four structures, where the type of the event determines which structure is pertinent:

- **mouse** is used for mouse events.
- **key** is used for keyboard events.
- **tracking** is for tracking-rectangle events.
- **compound** is for system-, kit-, and application-defined events.

NXPoint

DECLARED IN dpsclient/event.h

SYNOPSIS

```
typedef struct _NXPoint {
        NXCoord x;
        NXCoord y;
} NXPoint;
```

DESCRIPTION Represents a point in a Cartesian coordinate system.

NXSize

DECLARED IN dpsclient/event.h

SYNOPSIS

```
typedef struct _NXSize {
        NXCoord width;
        NXCoord height;
} NXSize;
```

DESCRIPTION Represents a two-dimensional size.

Symbolic Constants

All Contexts

DECLARED IN dpsclient/NSDPSContext.h

SYNOPSIS DPS_ALLCONTEXTS

DESCRIPTION This constant represents all extant contexts.

Alpha Constants

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS
NX_DATA
NX_ONES

DESCRIPTION These constants represent alpha values.

Character Set Values

DECLARED IN dpsclient/event.h

SYNOPSIS
NX_ASCIISET
NX_SYMBOLSET
NX_DINGBATSSET

DESCRIPTION These constants represent the values that may occur in the **data.key.charSet** field of an NXEvent structure.

Compositing Operations

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS

NX_CLEAR
NX_COPY
NX_SOVER
NX_SIN
NX_SOUT
NX_SATOP
NX_DOVER
NX_DIN
NX_DOUT
NX_DATOP
NX_XOR
NX_PLUSD
NX_HIGHLIGHT
NX_PLUSL

DESCRIPTION These represent the compositing operations used by **PScomposite()** and the NXImage class.

Error Code Bases

DECLARED IN dpsclient/dpsclient.h

SYNOPSIS

DPS_ERROR_BASE
DPS_NEXT_ERROR_BASE

DESCRIPTION These constants represent the lowest values for Display PostScript error codes.

Event Types

DECLARED IN dpsclient/event.h

Type	Meaning
NX_NULLEVENT	A non-event
NX_LMOUSEDOWN	Left mouse-down
NX_LMOUSEUP	Left mouse-up
NX_LMOUSEDRAGGED	left mouse-dragged
NX_MOUSEDOWN	Same as NX_LMOUSEDOWN
NX_MOUSEUP	Same as NX_LMOUSEUP
NX_MOUSEDRAGGED	Same as NX_LMOUSEDRAGGED
NX_RMOUSEDOWN	Right mouse-down
NX_RMOUSEUP	Right mouse-up
NX_RMOUSEDRAGGED	Right mouse-dragged
NX_MOUSEMOVED	Mouse-moved
NX_MOUSEENTERED	Mouse-entered
NX_MOUSEEXITED	Mouse-exited
NX_KEYDOWN	Key-down
NX_KEYUP	Key-up event
NX_FLAGSCHANGED	Flags-changed
NX_KITDEFINED	Application Kit-defined
NX_SYSDEFINED	System-defined
NX_APPDEFINED	Application-defined
NX_TIMER	Timer used for tracking
NX_CURSORUPDATE	Cursor tracking
NX_JOURNALEVENT	Event used by journaling
NX_FIRSTEVENT	The smallest-valued event constant
NX_LASTEVENT	The greatest-valued event constant
NX_ALLEVENTS	A value that includes all event types

DESCRIPTION These constants represent event types. They're passed as the **type** field of the NXEvent structure that's created when an event occurs.

Event Type Masks

DECLARED IN dpsclient/event.h

SYNOPSIS

```
NX_NULLEVENTMASK
NX_LMOUSEDOWNMASK
NX_LMOUSEUPMASK
NX_RMOUSEDOWNMASK
NX_RMOUSEUPMASK
NX_MOUSEMOVEDMASK
NX_LMOUSEDRAGGEDMASK
NX_RMOUSEDRAGGEDMASK
NX_MOUSEENTEREDMASK
NX_MOUSEEXITEDMASK
NX_KEYDOWNMASK
NX_KEYUPMASK
NX_FLAGSCHANGEDMASK
NX_KITDEFINEDMASK
NX_APPDEFINEDMASK
NX_SYSDEFINEDMASK
NX_TIMERMASK
NX_CURSORUPDATEMASK
NX_MOUSEDOWNMASK
NX_MOUSEUPMASK
NX_MOUSEDRAGGEDMASK
NX_JOURNALEVENTMASK
```

DESCRIPTION These masks correspond to the event types defined immediately above. They let you query the **type** field of an NXEvent structure for the existence of a particular event type.

Forever

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS NX_FOREVER

DESCRIPTION A long, long time. Typically used as the timeout argument to **DPSGetEvent()**.

Keyboard State Flags Masks

DECLARED IN dpsclient/event.h

Type	Meaning
NX_ALPHASHIFTMASK	Shift lock
NX_SHIFTMASK	Shift key
NX_CONTROLMASK	Control key
NX_ALTERNATEMASK	Alt key
NX_COMMANDMASK	Command key
NX_NUMERICPADMASK	Number pad key
NX_HELPMASK	Help key
NX_NEXTCTRLKEYMASK	Control key
NX_NEXTLSHIFTKEYMASK	Left shift key
NX_NEXTRSHIFTKEYMASK	Right shift key
NX_NEXTLCMDKEYMASK	Left command key
NX_NEXTRCMDKEYMASK	Right command key
NX_NEXTLALTKEYMASK	Left alt key
NX_NEXTRALTKEYMASK	Right alt key

DESCRIPTION These masks correspond to keyboard states that might be included in an NXEvent structure's **flags** mask. The masks are grouped as device-independent (NX_ALPHASHIFTMASK through NX_HELPMASK) and device-dependent (all others).

Miscellaneous Event Flags Masks

DECLARED IN dpsclient/event.h

Type	Meaning
NX_STYLUSPROXIMITYMASK	Stylus is in proximity (for tablets)
NX_NONCOALSESCEDMASK	Event coalescing disabled

DESCRIPTION These masks correspond to miscellaneous states that might be included in an NXEvent structure's **flags** mask.

Window Backing Types

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS

NX_RETAINED
NX_NONRETAINED
NX_BUFFERED

DESCRIPTION These represent the three backing types provided by window devices (and used by the Application Kit's Window objects).

Window Screen List Placement

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS

NX_ABOVE
NX_BELOW
NX_OUT

DESCRIPTION These represent the placement of a window device in the screen list.

Functions

This section describes functions and function-like macros available in the Application Kit library.

NSApplicationMain

SUMMARY This function is called by the **main** function to create and run the application.

DECLARED IN AppKit/NSApplication.h

SYNOPSIS int **NSApplicationMain(**int argc, const char *argv[]**)**

DESCRIPTION The **NSApplicationMain** function creates the application, loads the main nib file from the application's main bundle, and runs the application. You typically only call this function once, from your application's **main** function, which is usually generated automatically.

NSAvailableWindowDepths

SUMMARY This function returns the available NSWindowDepth values.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS const NSWindowDepth ***NSAvailableWindowDepths(**void**)**

DESCRIPTION **NSAvailableWindowDepths** returns a null-terminated array of NSWindowDepth values that specify which window depths are currently available. Window depth values are defined by the constants NSTwoBitGrayDepth, NSEightBitGrayDepth, and so on.

NSBeep

SUMMARY This function plays the system beep.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSBeep(**void**)**

DESCRIPTION This function plays the system beep. Users can select a sound to be played as the system beep. On a Macintosh, for example, you can change sounds with the Sound control panel.

NSBestDepth

SUMMARY This function attempts to return a window depth adequate for the specified parameters.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS NSWindowDepth **NSBestDepth(**NSString **colorSpace*, int *bps*, int *bpp*, BOOL *planar*, BOOL **exactMatch***)**

DESCRIPTION **NSBestDepth** returns a window depth deep enough for the given number of colors in *colorSpace*, bits per sample specified by *bps*, bits per pixel specified by *bpp*, and whether planar as specified by *planar*. Upon return, the variable pointed to by *exactMatch* is YES if the window depth can accommodate all of the values specified by the parameters, NO if it can't.

NSBitsPerPixelFromDepth

SUMMARY This function returns the bits per pixel for the specified window depth.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS int **NSBitsPerPixelFromDepth(**NSWindowDepth *depth***)**

DESCRIPTION **NSBitsPerPixelFromDepth** returns the number of bits per pixel for the window depth specified by *depth*.

SEE ALSO **NSBitsPerSampleFromDepth**

NSBitsPerSampleFromDepth

SUMMARY This function returns the bits per sample for the specified window depth.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS int **NSBitsPerSampleFromDepth(**NSWindowDepth *depth***)**

DESCRIPTION **NSBitsPerSampleFromDepth** returns the number of bits per sample (bits per pixel in each color component) for the window depth specified by *depth*.

SEE ALSO **NSBitsPerPixelFromDepth**

NSColorSpaceFromDepth

SUMMARY This function returns the name of the color space corresponding to the passed window depth.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS NSString ***NSColorSpaceFromDepth(**NSWindowDepth *depth***)**

DESCRIPTION Returns the color space name for the specified *depth*. For example, the returned color space name can be NSCalibratedRGBColorSpace, NSDeviceCMYKColorSpace, or so on.

NSConvertGlobalToWindowNumber

SUMMARY This function converts a global window number to a local window number.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSConvertGlobalToWindowNumber(**int *globalNum***,** unsigned int ***winNum***)**

DESCRIPTION In rare cases, two or more applications may need to refer to the same window. To pass a window number to another application, an application uses the global window number, which has been automatically assigned by the Window Server, rather than the local window number, which is assigned by the application.

An application uses the **NSConvertGlobalToWindowNumber** function to convert a window number from global to local. Given a global window number in *globalNum*, it returns the corresponding local window number in the location specified by *winNum*.

SEE ALSO **NSConvertWindowNumberToGlobal**

NSConvertGlyphsToPackedGlyphs

SUMMARY This function prepares a set of glyphs for processing by character-based routines.

DECLARED IN AppKit/NSFont.h

SYNOPSIS int **NSConvertGlyphsToPackedGlyphs**(NSGlyph **glBuf*, int *count*, NSMultibyteGlyphPacking *packing*, char **packedGlyphs*)

DESCRIPTION This function takes a buffer of glyphs, specified in the *glBuf* parameter, and packs them into a condensed character array. The character array is returned in the *packedGlyphs* parameter, which should have enough space for at least ((4 * *count*) + 1) bytes to guarantee that the packed glyphs fit. The *count* parameter specifies the number of glyphs in *glBuf*. The *packing* parameter specifies how the glyphs are currently packed.

NSConvertWindowNumberToGlobal

SUMMARY This function converts a local window number to a global window number.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSConvertWindowNumberToGlobal(**int *winNum*, unsigned int **globalNum***)*

DESCRIPTION In rare cases, two or more applications may need to refer to the same window. To pass a window number to another application, an application uses the global window number, which has been automatically assigned by the Window Server, rather than the local window number, which is assigned by the application.

NSConvertWindowNumberToGlobal takes the local window number and places the corresponding global window number in the variable specified by *globalNum*. This global number can then be passed to other applications that need access to the window.

SEE ALSO **NSConvertGlobalToWindowNumber**

NSCopyBitmapFromGState

SUMMARY This function copies a bitmap image to a destination rectangle.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSCopyBitmapFromGState(**int *srcGState***,** NSRect *srcRect***,** NSRect *destRect***)**

DESCRIPTION This function copies the pixels in the rectangle *srcRect* to the rectangle *destRect*. The source rectangle is defined in the graphics state designated by *srcGState*. The destination is defined in the current graphics state.

SEE ALSO **NSCopyBits**

NSCopyBits

SUMMARY This function copies a bitmap image to the location specified by a destination point.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSCopyBits(**int *srcGState***,** NSRect *srcRect***,** NSPoint *destPoint***)**

DESCRIPTION This function copies the pixels in the rectangle specified by *srcRect* to the location specified by *destPoint*. The source rectangle is defined in the graphics state designated by *srcGState*. If *srcGState* is NSNullObject, the current graphics state is assumed. The *destPoint* destination is defined in the current graphics state.

SEE ALSO **NSCopyBitmapFromGState**

NSCountWindows

SUMMARY This function counts the number of on-screen windows belonging to an application.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSCountWindows(**int **count***)**

DESCRIPTION **NSCountWindows** counts the number of on-screen windows belonging to the application; it returns the number by reference in the *count* parameter.

SEE ALSO **NSWindowList**

NSCreateFileContentsPboardType

SUMMARY This function returns a pasteboard type based on the passed file type.

DECLARED IN AppKit/NSPasteboard.h

SYNOPSIS NSString ***NSCreateFileContentsPboardType(**NSString ****fileType*)**

DESCRIPTION **NSCreateFileContentsPboardType** returns an NSString to a pasteboard type representing a file's contents based on the supplied string *fileType*. *fileType* should generally be the extension part of a file name. The conversion from a named file type to a pasteboard type is simple; no mapping to standard pasteboard types is attempted.

SEE ALSO **NSCreateFilenamePboardType, NSGetFileType, NSGetFileTypes**

NSCreateFilenamePboardType

SUMMARY This function returns a pasteboard type based on the passed file type.

DECLARED IN AppKit/NSPasteboard.h

SYNOPSIS NSString ***NSCreateFilenamePboardType(**NSString ****fileType*)**

DESCRIPTION **NSCreateFilenamePboardType** returns an NSString to a pasteboard type representing a file name based on the supplied string *fileType*.

SEE ALSO **NSCreateFileContentsPboardType, NSGetFileType, NSGetFileTypes**

NSDottedFrameRect

SUMMARY << Description forthcoming >>

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSDottedFrameRect**(NSRect *aRect*)

DESCRIPTION << Description forthcoming >>

NSDrawBitmap

SUMMARY This function draws a bitmap image.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSDrawBitmap(**const NSRect *rect*, int *pixelsWide*, int *pixelsHigh*, int *bitsPerSample*, int *samplesPerPixel*, int *bitsPerPixel*, int *bytesPerRow*, BOOL *isPlanar*, BOOL *hasAlpha*, NSColorSpace *colorSpace*, const unsigned char *const *data*[5]**)**

Warning: This function is marginally obsolete. Most applications are better served using the NSBitmapImageRep class to read and display bitmap images.

DESCRIPTION The **NSDrawBitmap** function renders an image from a bitmap, binary data that describes the pixel values for the image (this function replaces **NSImageBitmap**).

NSDrawBitmap renders a bitmap image using an appropriate PostScript operator—**image**, **colorimage**, or **alphaimage**. It puts the image in the rectangular area specified by its first argument, *rect*; the rectangle is specified in the current coordinate system and is located in the current window. The next two arguments, *pixelsWide* and *pixelsHigh*, give the width and height of the image in pixels. If either of these dimensions is larger or smaller than the corresponding dimension of the destination rectangle, the image will be scaled to fit.

The remaining arguments to **NSDrawBitmap** describe the bitmap data, as explained in the following paragraphs.

bitsPerSample is the number of bits per sample for each pixel and *samplesPerPixel* is the number of samples per pixel. *bitsPerPixel* is based on *samplesPerPixel* and the configuration of the

bitmap: if the configuration is planar, then the value of *bitsPerPixel* should equal the value of *bitsPerSample*; if the configuration isn't planar (is meshed instead), *bitsPerPixel* should equal *bitsPerSample * samplesPerPixel*.

bytesPerRow is calculated in one of two ways, depending on the configuration of the image data (data configuration is described below). If the data is planar, *bytesPerRow* is (7 + (*pixelsWide * bitsPerSample*)) / 8. If the data is meshed, *bytesPerRow* is (7 + (*pixelsWide * bitsPerSample * samplesPerPixel*)) / 8.

A sample is data that describes one component of a pixel. In an RGB color system, the red, green, and blue components of a color are specified as separate samples, as are the cyan, magenta, yellow, and black components in a CMYK system. Color values in a gray scale are a single sample. Alpha values that determine transparency and opaqueness are specified as a coverage sample separate from color. In bitmap images with alpha, the color (or gray) components have to be premultiplied with the alpha. This is the way images with alpha are displayed, this is the way they are read back, and this is the way they are stored in TIFFs.

isPlanar refers to the way data is configured in the bitmap. This flag should be set YES if a separate data channel is used for each sample. The function provides for up to five channels, *data1*, *data2*, *data3*, *data4*, and *data5*. It should be set NO if sample values are interwoven in a single channel (meshed); all values for one pixel are specified before values for the next pixel.

Figure 1 illustrates these two ways of configuring data.

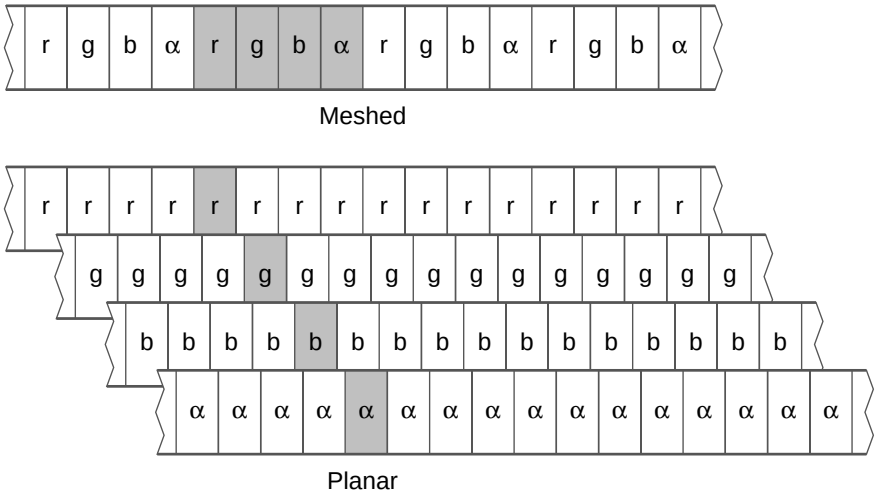


Figure 1. Planar and Meshed Configurations

As shown in the illustration, color samples (rgb) precede the coverage sample (α) in both configurations.

Gray-scale windows store pixel data in planar configuration; color windows store it in meshed configuration. **NSDrawBitmap** can render meshed data in a planar window, or planar data in a meshed window. However, it's more efficient if the image has a depth (*bitsPerSample*) and configuration (*isPlanar*) that matches the window.

hasAlpha indicates whether the image contains alpha. If it does, the number of samples should be 1 greater than the number of color components in the model (e.g., 4 for RGB).

colorSpace can be NS_CustomColorSpace, indicating that the image data is to be interpreted according to the current color space in the PostScript graphics state. This allows for imaging using custom color spaces. The image parameters supplied as the other arguments should match what the color space is expecting.

If the image data is planar, *data*[0] through *data*[*samplesPerPixel*-1] point to the planes; if the data is meshed, only *data*[0] needs to be set.

NSDrawButton, NSDrawGrayBezel, NSDrawGroove, NSDrawTiledRects, NSDrawWhiteBezel, NSFrameRect, NSFrameRectWithWidth

SUMMARY Draw a bordered rectangle

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS

void **NSDrawButton(**const NSRect *aRect*, const NSRect *clipRect***)**
void **NSDrawGrayBezel(**const NSRect *aRect*, const NSRect *clipRect***)**
void **NSDrawGroove(**const NSRect *aRect*, const NSRect *clipRect***)**
void **NSDrawWhiteBezel(**const NSRect *aRect*, const NSRect *clipRect***)**
NSRect **NSDrawTiledRects(**NSRect *aRect*, const NSRect *clipRect*, const NSRectEdge **sides*, const float **grays*, int *count***)**
void **NSFrameRect(**const NSRect *aRect***)**
void **NSFrameRectWithWidth(**const NSRect *aRect*, NSCoord *frameWidth***)**

DESCRIPTION These functions draw rectangles with borders. **NSDrawButton** draws the rectangle used to signify a user-interface button, **NSDrawTiledRects** is a generic function that can be used to draw different types of borders, and the other functions provide ready-made bezeled, grooved, or line

borders. These borders can be used to outline an area or to give rectangles the effect of being recessed from or elevated above the surface of the screen, as shown in Figure 2.

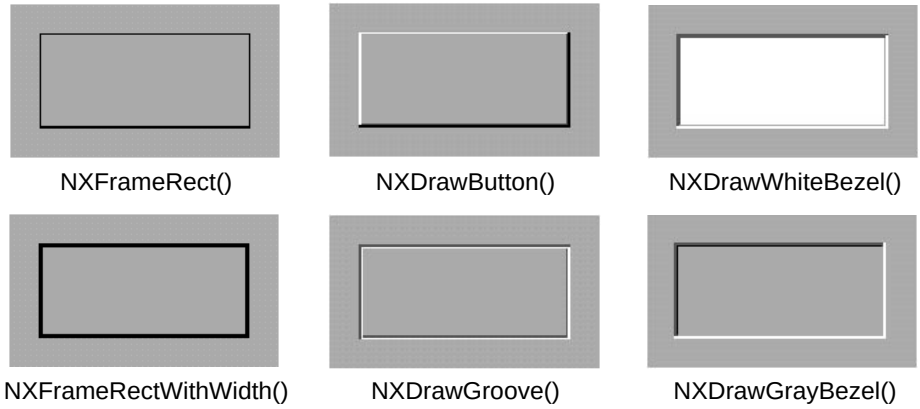


Figure 2. Rectangle Borders

Each function's first argument specifies the rectangle within which the border is to be drawn in the current coordinate system. Since these functions are often used to draw the border of a View, this rectangle will typically be that View's bounds rectangle. Some of the functions also take a clipping rectangle; only those parts of *aRect* that lie within the clipping rectangle will be drawn.

As its name suggests, **NSDrawWhiteBezel** fills in its rectangle with white; **NSDrawButton**, **NSDrawGrayBezel**, and **NSDrawGroove** use light gray. These functions are designed for rectangles that are defined in unscaled, unrotated coordinate systems (that is, where the y-axis is vertical, the x-axis is horizontal, and a unit along either axis is equal to one screen pixel). The coordinate system can be either flipped or unflipped. The sides of the rectangle should lie on pixel boundaries.

NSFrameRect and **NSFrameRectWithWidth** draw a frame around the inside of a rectangle in the current color. **NSFrameRect** draws a frame with a width equal to 1.0 in the current coordinate system; **NSFrameRectWithWidth** allows you to set the width of the frame. Since the frame is drawn inside the rectangle, it will be visible even if drawing is clipped to the rectangle (as it would be if the rectangle were a View object). These functions work best if the sides of the rectangle lie on pixel boundaries.

In addition to its *aRect* and *clipRect* arguments, **NSDrawTiledRects** takes three more arguments, which determine how thick the border is and what gray levels are used to form it. **NSDrawTiledRects** works through the Foundation framework's **NSDivideRect** function to take successive 1.0–unit-wide slices from the sides of the rectangle specified by the *sides* argument. Each slice is then drawn using the corresponding gray level from *grays*. **NSDrawTiledRects**

makes and draws these slices *count* number of times. **NSDivideRect** returns a pointer to the rectangle after the slice has been removed; therefore, if a side is used more than once, the second slice is made inside the first. This also makes it easy to fill in the rectangle inside of the border.

In the following example, **NSDrawTiledRects** draws a bezeled border consisting of a 1.0–unit-wide white line at the top and on the left side, and a 1.0–unit-wide dark-gray line inside a 1.0–unit-wide black line on the other two sides. The rectangle inside this border is filled in using light gray.

```
int    mySides[] = {NX_YMIN, NX_XMAX, NX_YMAX, NX_XMIN,
                    _YMIN, NX_XMAX};
float  myGrays[] = {NS_BLACK, NS_BLACK, NS_WHITE, NS_WHITE,
                    NS_DARKGRAY, NS_DARKGRAY};
NSRect *aRect;

NSDrawTiledRects(aRect, (NSRect *)0, mySides, myGrays, 6);
PSsetgray(NS_LIGHTGRAY);
PSrectfill(aRect->origin.x, aRect->origin.y,
          aRect->size.width, aRect->size.height);
```

As shown, **mySides** is an array that specifies sides of a rectangle; for example, NX_YMIN selects the side parallel to the x-axis with the smallest y-coordinate value. **myGrays** is an array that specifies the successive gray levels to be used in drawing parts of the border.

NSDrawTiledRects returns a pointer to the rectangle that lies within the border.

NSDrawColorTiledRects

SUMMARY << Description forthcoming >>

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS NSRect **NSDrawColorTiledRects**(NSRect *boundsRect*, NSRect *clipRect*, const NSRectEdge **sides*, NSColor ***colors*, int *count*)

DESCRIPTION << Description forthcoming >>

NSDrawDarkBezel

SUMMARY << Description forthcoming >>

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSDrawDarkBezel**(NSRect *aRect*, NSRect *clipRect*)

DESCRIPTION << Description forthcoming >>

NSDrawLightBezel

SUMMARY << Description forthcoming >>

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSDrawLightBezel**(NSRect *aRect*, NSRect *clipRect*)

DESCRIPTION << Description forthcoming >>

NSEraseRect

SUMMARY This function erases the passed rect by filling it with white.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSEraseRect(**const NSRect *aRect***)**

DESCRIPTION As its name suggests, **NSEraseRect** erases the rectangle referred to by its argument, filling it with white. It does not alter the current color.

SEE ALSO **NSHighlightRect, NSRectClip, NSRectClipList, NSRectFill, NSRectFillList, NSRectFillListWithColors, NSRectFillListWithGrays, NSUnionRect (Foundation Kit)**

NSEventMaskFromType

SUMMARY This function returns the event mask for the specified type.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS unsigned int **NSEventMaskFromType(**NSEventType *type***)**

DESCRIPTION **NSEventMaskFromType** returns the event mask corresponding to the specified *type* (an enumerated constant). The returned mask is equal to 1 left-shifted by *type* bits.

NSFrameLinkRect

SUMMARY This function draws a distinctive outline around linked data.

DECLARED IN AppKit/NSDataLinkManager.h

SYNOPSIS void **NSFrameLinkRect(**NSRect *aRect*, BOOL *isDestination***)**

DESCRIPTION **NSFrameLinkRect** draws a distinctive link outline just outside the rectangle specified by *aRect*. To draw an outline around a destination link, *isDestination* should be YES, otherwise it should be NO.

SEE ALSO **NSLinkFrameThickness**

NSGetAlertPanel

SUMMARY This function returns an alert panel.

DECLARED IN AppKit/NSPanel.h

SYNOPSIS id **NSGetAlertPanel(**NSString **title*, NSString **msg*, NSString **defaultButton*, NSString **alternateButton*, NSString **otherButton*, ...**)**

DESCRIPTION **NSGetAlertPanel** returns an NSAlert panel that can be used to set up a modal session. A modal session is useful for allowing the user to interrupt the program. During a modal session, you can perform activities while the panel is displayed and check at various points in your program whether the user has clicked one of the panel's buttons. The arguments for this function are the same as those for the **NSRunAlertPanel** function, but unlike that function, no button is displayed if *defaultButton* is **nil**.

To set up a modal session, send the Application object a **beginModalSession:for:** message with the panel returned by **NSGetAlertPanel** as its second argument. When you want to check if the user has clicked one of the panel's buttons, use **runModalSession:**. To end the modal session, use **endModalSession:**. When you're finished with the panel created by **NSGetAlertPanel**, you must free it by passing it to **NSReleaseAlertPanel**.

SEE ALSO **NSGetCriticalAlertPanel, NSGetInformationalAlertPanel, NSReleaseAlertPanel, NSRunAlertPanel, NSRunCriticalAlertPanel, NSRunInformationalAlertPanel**

NSGetCriticalAlertPanel

SUMMARY This function returns an alert panel to display a critical message.

DECLARED IN AppKit/NSPanel.h

SYNOPSIS id **NSGetCriticalAlertPanel(**NSString **title*, NSString **msg*, NSString **defaultButton*, NSString **alternateButton*, NSString **otherButton*, ...**)**

DESCRIPTION **NSGetCriticalAlertPanel** returns an NSAlert panel that can be used to set up a modal session. Unlike the **NSRunCriticalAlertPanel** function, no button is displayed if *defaultButton* is **nil**. When you're finished with the panel created by **NSGetCriticalAlertPanel**, you must free it by passing it to **NSReleaseAlertPanel**.

The arguments for this function are the same as those for the **NSRunAlertPanel** function. For more information on using a panel in a modal session, see **NSGetAlertPanel**.

SEE ALSO **NSGetAlertPanel, NSGetInformationalAlertPanel, NSReleaseAlertPanel, NSRunAlertPanel, NSRunCriticalAlertPanel, NSRunInformationalAlertPanel**

NSGetFileType

SUMMARY This function returns a file type based on the passed pasteboard type.

DECLARED IN AppKit/NSPasteboard.h

SYNOPSIS NSString ***NSGetFileType(**NSArray ***pboardType*)

DESCRIPTION **NSGetFileType** is the inverse of both **NSCreateFileContentsPboardType** and **NSCreateFilenamePboardType**. When passed a pasteboard type as returned by those functions, it returns the extension or file name from which the type was derived. It returns **nil** if *pboardType* isn't a pasteboard type created by those functions.

SEE ALSO **NSCreateFileContentsPboardType, NSCreateFilenamePboardType, NSGetFileTypes**

NSGetFileTypes

SUMMARY This function returns an array of file type based on the passed pasteboard types.

DECLARED IN AppKit/NSPasteboard.h

SYNOPSIS NSArray ***NSGetFileTypes(**NSArray ***pboardType*)

DESCRIPTION **NSGetFileTypes** accepts a null-terminated array of pointers to pasteboard types and returns a null-terminated array of the unique extensions and file names from the file-content and file-name types found in the input array. It returns **nil** if the input array contains no file-content or file-name types. The returned array is allocated and must be freed by the caller. The pointers in the return array point into strings passed in the input array.

SEE ALSO **NSCreateFileContentsPboardType, NSCreateFilenamePboardType, NSGetFileType**

NSGetInformationalAlertPanel

SUMMARY This function returns an alert panel to display an informational message.

DECLARED IN AppKit/NSPanel.h

SYNOPSIS id **NSGetInformationalAlertPanel(**NSString **title*, NSString **msg*, NSString **defaultButton*, NSString **alternateButton*, NSString **otherButton*, ...**)**

DESCRIPTION **NSGetInformationalAlertPanel** returns an NSAlert panel that can be used to set up a modal session. Unlike the **NSRunInformationalAlertPanel** function, no button is displayed if *defaultButton* is **nil**. When you're finished with the panel created by **NSGetInformationalAlertPanel**, you must free it by passing it to **NSReleaseAlertPanel**.

The arguments for this function are the same as those for the **NSRunAlertPanel** function. For more information on using a panel in a modal session, see **NSGetAlertPanel**.

SEE ALSO **NSGetAlertPanel, NSGetCriticalAlertPanel, NSReleaseAlertPanel, NSRunAlertPanel, NSRunCriticalAlertPanel, NSRunInformationalAlertPanel**

NSGetWindowServerMemory

SUMMARY This function returns the amount of memory being used by a context.

DECLARED IN AppKit/Application.h

SYNOPSIS int **NSGetWindowServerMemory(**DPSContext *context*, int **virtualMemory*, int **windowBackingMemory*, NSString ***windowDumpStream***)**

DESCRIPTION **NSGetWindowServerMemory** calculates the amount of Window Server memory being used at the moment by the given Window Server context. If **nil** is passed for the context, the current context is used. The amount of PostScript virtual memory used by the current context is returned in the **int** pointed to by *virtualMemory*; the amount of window backing store used by windows owned by the current context is returned in the **int** pointed to by *windowBackingMemory*. The sum of these two numbers is the amount of the Window Server's memory that this context is responsible for.

To calculate these numbers, **NSGetWindowServerMemory** uses the PostScript language operators **dumpwindows** and **vmstatus**. It takes some time to execute; thus, calling this function in normal operation is not recommended.

If a non–nil value is passed in for *windowDumpStream*, the information returned from the **dumpwindows** operator is echoed to the specified stream. This can be useful for finding out more about which windows are using up your storage.

Normally, **NSGetWindowServerMemory** returns 0. If **nil** is passed for context and there's no current DPS context, this function returns –1.

NSHighlightRect

SUMMARY This function highlights the passed rect by filling it with white.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSHighlightRect(**const NSRect *aRect***)**

DESCRIPTION **NSHighlightRect** uses the **compositerect** operator to highlight the rectangle referred to by its argument. Light gray becomes white, and white becomes light gray. This function must be called twice, once to highlight the rectangle and once to unhighlight it; the rectangle should not be left in its highlighted state. When not drawing on the screen, the compositing operation is replaced by one that fills the rectangle with light gray.

SEE ALSO **NSHighlightRect, NSRectClip, NSRectClipList, NSRectFill, NSRectFillList, NSRectFillListWithColors, NSRectFillListWithGrays, NSUnionRect (Foundation Kit)**

NSInterfaceStyleForKey

SUMMARY This function returns an interface style value for the specified key and responder.

DECLARED IN AppKit/NSInterfaceStyle.h

SYNOPSIS NSInterfaceStyle **NSInterfaceStyleForKey(**NSString **key*, NSResponder **responder***)**

DESCRIPTION You call the **NSInterfaceStyleForKey** function to determine an interface style based on a key and a responder, either of which may be **nil**. An NSInterfaceStyle value specifies the style in which an interface item, such as a button or a scrollbar, should be drawn. For example, a value of NSMacintoshInterfaceStyle indicates an item should be drawn in the Macintosh style. The enum values defined for NSInterfaceStyle are NSNoInterfaceStyle, NSNextStepInterfaceStyle, NSWindows95InterfaceStyle, and NSMacintoshInterfaceStyle. Note that **NSInterfaceStyleForKey** never returns NSNoInterfaceStyle.

The interface style value returned by **NSInterfaceStyleForKey** depends on several factors. If *responder* is not **nil** and if *responder* specifies an interface style other than NSNoInterfaceStyle, **NSInterfaceStyleForKey** returns the responder's style, and *key* is ignored.

Otherwise, if *key* is not **nil** and there is an interface style for *key* specified by the defaults system, **NSInterfaceStyleForKey** returns the interface style for *key* from the defaults system.

Finally, if *key* is **nil**, or if there is no interface style for *key* specified by the defaults system, **NSInterfaceStyleForKey** returns the global interface style specified by the defaults system.

The defaults system allows an application to customize its behavior to match a user's preferences.You can read about the defaults system in the documentation for NSUserDefaults.

NSLinkFrameThickness

SUMMARY This function returns the thickness of the outline around linked data.

DECLARED IN AppKit/NSDataLinkManager.h

SYNOPSIS float **NSLinkFrameThickness(**void**)**

DESCRIPTION **NSLinkFrameThickness** returns the thickness of the link outline around linked data so that the outline can be properly erased by the application, or for other purposes.

SEE ALSO **NSFrameLinkRect**

NSNumberOfColorComponents

SUMMARY This function returns the number of color components in the specified color space.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS int **NSNumberOfColorComponents(**NSString **colorSpaceName***)*

DESCRIPTION **NSNumberOfColorComponents** returns the number of color components in the color space whose name is provided by *colorSpaceName*.

NSPerformService

SUMMARY This function programmatically invokes a Services menu service.

DECLARED IN AppKit/Listener.h

SYNOPSIS BOOL **NSPerformService(**NSString **itemName*, NSPasteboard **pboard*)

DESCRIPTION **NSPerformService** allows an application to programmatically invoke a service found in its services menu. *itemName* is a Services menu item, in any language. If the requested service is from a submenu of the Services menu, *itemName* must contain a slash (for example, “Mail/Selection”). The Pasteboard *pboard* must contain the data required by the service, and when the function returns, *pboard* will contain the data supplied by the service provider.

NSPerformService returns YES if the service is successfully performed, NO otherwise.

NSPlanarFromDepth

SUMMARY This function returns whether the specified window depth is planar.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS BOOL **NSPlanarFromDepth(**NSWindowDepth *depth***)**

DESCRIPTION **NSPlanarFromDepth** returns YES if the specified window depth is planar and NO if it is not.

NSReadPixel

SUMMARY This function reads a pixel value at the specified location.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS **NSColor *NSReadPixel(NSPoint** *passedPoint***)**

DESCRIPTION **NSReadPixel** returns the color of the pixel at the given location. The *location* argument is taken in the current coordinate system—in other words, you must lock focus on the View that contains the pixel that you wish to query, and then pass the coordinate for the pixel in the View's coordinate system.

NSRectClip

SUMMARY This function modifies the current clipping path by intersecting it with the passed rect.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSRectClip(**NSRect *aRect***)**

DESCRIPTION **NSRectClip** intersects the current clipping path with the rectangle referred to by its argument, *aRect*, to determine a new clipping path. This function works through the **rectclip** operator. After computing the new clipping path, the current path is reset to empty.

SEE ALSO **NSEraseRect, NSHighlightRect, NSRectClipList, NSRectFill, NSRectFillList, NSRectFillListWithColors, NSRectFillListWithGrays, NSUnionRect (Foundation Kit)**

NSRectClipList

SUMMARY This function modifies the current clipping path by intersecting it with the passed rect.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSRectClipList(**const NSRect **rects*, int *count***)**

DESCRIPTION **NSRectClipList** takes an array of *count* number of rectangles, constructs a path that's the graphic union of those rectangles, and intersects that path with the current clipping path. This function works through the **rectclip** operator. After computing the new clipping path, the current path is reset to empty.

SEE ALSO **NSEraseRect, NSHighlightRect, NSRectClip, NSRectFill, NSRectFillList, NSRectFillListWithColors, NSRectFillListWithGrays, NSUnionRect (Foundation Kit)**

NSRectFill

SUMMARY This function fills the passed rect with the current color.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSRectFillList(**const NSRect *aRect***)**

DESCRIPTION **NSRectFill** fills the rectangle referred to by its argument with the current color. It works through the **rectfill** operator.

SEE ALSO **NSEraseRect, NSHighlightRect, NSRectClip, NSRectClipList, NSRectFillList, NSRectFillListWithColors, NSRectFillListWithGrays, NSUnionRect (Foundation Kit)**

NSRectFillList

SUMMARY This function fills the rectangles in the passed list with the current color.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSRectFillList(**const NSRect **rects*, int *count***)**

DESCRIPTION **NSRectFillList** fills a list of *count* rectangles with the current color. It works through the **rectfill** operator.

SEE ALSO **NSEraseRect, NSHighlightRect, NSRectClip, NSRectClipList, NSRectFill, NSRectFillListWithColors, NSRectFillListWithGrays, NSUnionRect (Foundation Kit)**

NSRectFillListWithColors

SUMMARY This function fills the rectangles in the passed list with the current color.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSRectFillListWithColors(**const NSRect **rects*, NSColor ***colors*, int *count***)**

DESCRIPTION **NSRectFillListWithColors** takes a list of *count* rectangles and a matching list of *count* color values. The first rectangle is filled with the first color, the second rectangle with the second color, and so on. There must be an equal number of rectangles and color values. The rectangles should

not overlap; the order in which they'll be filled can't be guaranteed. This function alters the current color of the current graphics state, setting it unpredictably to one of the values passed in *colors*.

SEE ALSO **NSEraseRect, NSHighlightRect, NSRectClip, NSRectClipList, NSRectFill, NSRectFillList, NSRectFillListWithGrays, NSUnionRect (Foundation Kit)**

NSRectFillListWithGrays

SUMMARY This function fills the rectangles in the passed list with the current color.

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSRectFillListWithGrays(**const NSRect **rects*, const float **grays*, int *count***)**

DESCRIPTION **NSRectFillListWithGrays** takes a list of *count* rectangles and a matching list of *count* gray values. The first rectangle is filled with the first gray, the second rectangle with the second gray, and so on. There must be an equal number of rectangles and gray values. The rectangles should not overlap; the order in which they'll be filled can't be guaranteed. This function alters the current color of the current graphics state, setting it unpredictably to one of the values passed in *grays*.

SEE ALSO **NSEraseRect, NSHighlightRect, NSRectClip, NSRectClipList, NSRectFill, NSRectFillList, NSRectFillListWithColors, NSUnionRect (Foundation Kit)**

NSRegisterServicesProvider

SUMMARY This function registers a service provider.

DECLARED IN AppKit/NSApplication.h

SYNOPSIS void **NSRegisterServicesProvider(**id *provider*, NSString **name***)**

DESCRIPTION **NSRegisterServicesProvider** registers *provider* as a service provider and associates it with the specified *name*. *name* should be unique; it is the name by which the service is advertised to service requestors.

NSApplications shouldn't use this function. Instead, they should use NSApplication's **setServicesProvider:** method, passing a non-nil argument.

SEE ALSO **NSSetShowsServicesMenuItem, NSShowsServicesMenuItem, NSUnRegisterServicesProvider,**

NSReleaseAlertPanel

SUMMARY This function releases an attention panel.

DECLARED IN AppKit/NSPanel.h

SYNOPSIS void **NSReleaseAlertPanel(**id *alertPanel***)**

DESCRIPTION When you're finished with a panel created by a function such as **NSGetAlertPanel**, **NSGetCriticalAlertPanel**, or **NSGetInformationalAlertPanel**, you must free it by passing it to **NSReleaseAlertPanel**.

SEE ALSO **NSGetAlertPanel, NSRunAlertPanel, NSRunCriticalAlertPanel**

NSRunAlertPanel

SUMMARY This function creates an attention panel.

DECLARED IN AppKit/NSPanel.h

SYNOPSIS int **NSRunAlertPanel(**NSString **title*, NSString **msg*, NSString **defaultButton*, NSString **alternateButton*, NSString **otherButton*, ...**)**

DESCRIPTION **NSRunAlertPanel** creates an attention panel that alerts the user to some consequence of a requested action; the panel may also let the user cancel or modify the action. **NSRunAlertPanel** runs the panel in a modal event loop.

The first argument is the title of the panel, which should be at most a few words long. The default title is "Alert". The next argument is the message that's displayed in the panel. It can use

printf-style formatting characters; any necessary arguments should be listed at the end of the function's argument list (after the *otherButton* argument). For more information on formatting characters, see the Rhapsody man page for **printf**.

There are arguments to supply titles for up to three buttons, which will be displayed in a row across the bottom of the panel. The panel created by **NSRunAlertPanel** must have at least one button, which will have the symbol for the Return key; if you pass a nil title to the other two buttons, they won't be created. If **nil** is passed as the *defaultButton*, "OK" will be used as its title.

NSRunAlertPanel not only creates the panel, it puts the panel on screen and runs it using the **runModalFor:** method defined in the Application class. This method sets up a modal event loop that causes the panel to remain on screen until the user clicks one of its buttons. **NSRunAlertPanel** then removes the panel from the screen list and returns a value that indicates which of the three buttons the user clicked: NS_ALERTDEFAULT, NS_ALERTALTERNATE, or NS_ALERTOTHER. (If an error occurred while creating the panel, NS_ALERTERROR is returned.) For efficiency, **NSRunAlertPanel** creates the panel the first time it's called and reuses it on subsequent calls, reconfiguring it if necessary.

SEE ALSO **NSGetAlertPanel, NSGetCriticalAlertPanel, NSGetInformationalAlertPanel, NSReleaseAlertPanel, NSRunCriticalAlertPanel, NSRunInformationalAlertPanel**

NSRunCriticalAlertPanel

SUMMARY This function creates and runs a critical attention panel.

DECLARED IN AppKit/NSPanel.h

SYNOPSIS int **NSRunCriticalAlertPanel(**NSString **title*, NSString **msg*, NSString **defaultButton*, NSString **alternateButton*, NSString **otherButton*, ...**)**

DESCRIPTION **NSRunCriticalAlertPanel** creates an attention panel that alerts the user to some critical consequence of a requested action; the panel lets the user cancel the action and may allow the user to modify the action. It then runs the panel in a modal event loop.

The arguments for this function are the same as those for the **NSRunAlertPanel** function.

SEE ALSO **NSGetAlertPanel, NSGetCriticalAlertPanel, NSGetInformationalAlertPanel, NSReleaseAlertPanel, NSRunAlertPanel, NSRunInformationalAlertPanel**

NSRunInformationalAlertPanel

SUMMARY This function creates and runs an informational attention panel.

DECLARED IN AppKit/NSPanel.h

SYNOPSIS int **NSRunInformationalAlertPanel(**NSString **title*, NSString **msg*, NSString **defaultButton*, NSString **alternateButton*, NSString **otherButton*, ...**)**

DESCRIPTION **NSRunInformationalAlertPanel** creates an informational attention panel that provides information related to a requested action. It then runs the panel in a modal event loop.

The arguments for this function are the same as those for the **NSRunAlertPanel** function.

SEE ALSO **NSGetAlertPanel, NSGetCriticalAlertPanel, NSGetInformationalAlertPanel, NSReleaseAlertPanel, NSRunAlertPanel, NSRunCriticalAlertPanel**

NSSetShowsServicesMenuItem

SUMMARY This function specifies whether an item should be included in Services menus.

DECLARED IN AppKit/NSApplication.h

SYNOPSIS int **NSSetShowsServicesMenuItem(**NSString * *itemName*, BOOL *enabled***)**

DESCRIPTION **NSSetShowsServicesMenuItem** is used by a service-providing application to specify whether the Services menus of other applications will contain the *itemName* command; if so, users of those applications will be able to request services through that command. If *enabled* is YES, the Application Kit will build Services menus for other applications that include the *itemName* command. If *enabled* is NO, *item* won't appear in any application's Services menu. *itemName* should be the same, language-independent character string entered in the "Menu Item:" field of the services file.

Service-providing applications should let users decide whether the Services menus of other applications they use should include the *itemName* command.

NSSetShowsServicesMenuItem returns 0 if it's successful in enabling or disabling the *itemName* command, and a number other than 0 if not.

SEE ALSO **NSRegisterServicesProvider, NSShowsServicesMenuItem, NSUnRegisterServicesProvider,**

NSShowsServicesMenuItem

SUMMARY This function specifies whether a Services menu item is currently enabled.

DECLARED IN AppKit/NSApplication.h

SYNOPSIS BOOL **NSShowsServicesMenuItem(**NSString * *itemName***)**

DESCRIPTION **NSShowsServicesMenuItem** returns YES if *itemName* is currently enabled, and NO if it's not. *itemName* should be the same, language-independent character string entered in the "Menu Item: " field of the services file.

SEE ALSO **NSRegisterServicesProvider, NSSetShowsServicesMenuItem, NSUnRegisterServicesProvider,**

NSUnRegisterServicesProvider

SUMMARY This function unregisters a service provider.

DECLARED IN AppKit/NSApplication.h

SYNOPSIS void **NSUnRegisterServicesProvider(**NSString **name***name***)**

DESCRIPTION **NSUnRegisterServicesProvider** unregisters the object named by *name* as a service provider.

NSApplications shouldn't use this function. Instead, they should use NSApplication's **setServicesProvider:** method, passing a **nil** argument.

SEE ALSO **NSRegisterServicesProvider, NSSetShowsServicesMenuItem, NSShowsServicesMenuItem**

NSUpdateDynamicServices

SUMMARY This function causes the services information for the system to be updated.

DECLARED IN AppKit/NSApplication.h

SYNOPSIS void **NSUpdateDynamicServices(**void**)**

DESCRIPTION **NSUpdateDynamicServices** is used by a service-providing application to re-register the services it's willing to provide. To do this, you create a file with the extension ".service" and place it in the application's path or **~/Library/Services**. The content of the file is identical to a normal service file (see the "Other Features" section for a description of service file format). You then call this function.

It is only necessary to call **NSUpdateDynamicServices** if your program adds dynamic services to the system.

NSWindowList

SUMMARY Get information about an application's windows

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS void **NSWindowList(**int *size*, int *list*[]**)**

DESCRIPTION **NSWindowList** provides an ordered list of the application's on-screen windows. It fills the *list* array with up to *size* window numbers; the order of windows in the array is the same as their order in the Window Server's screen list (their front-to-back order on the screen). Use the count obtained by **NSCountWindows** to specify the size of the array for **NSWindowList**.

SEE ALSO **NSCountWindows**

Client Library Functions

Note: This section has not been updated and has not received recent technical review. It is included in this release to test the linkage between application development tools and on-line documentation. The information in this section should be considered at best preliminary and subject to change. An updated version of this file will be included in the next release.

DPSAddFD

SUMMARY Monitor a file descriptor

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSAddFD(**int *fd*, DPSFDProc *handler*, void **userData*, int *priority***)**

DESCRIPTION **DPSAddFD** registers the function *handler* to be called each time there is activity with the file specified by file descriptor *fd*. The function is called provided the following are true:

- The file descriptor *fd* must be valid and open; typically *fd* is generated through a call to **open**. There needn't be any data waiting to be read on *fd*.
- *priority*, an integer from 0 to 30, must be equal to or greater than the application's current priority threshold. See **DPSAddTimedEntry** for a further explanation.

DPSFDProc, *handler*'s defined type, takes the form

```
void *handler(int fd, void *userData)
```

where *fd* is the file descriptor that prompted the function call and *userData* is the same pointer that was passed as the third argument to **DPSAddFD**. The *userData* pointer is provided as a convenience, allowing you to pass arbitrary data to *handler*.

Typically, **DPSAddFD** is used to listen to a socket or pipe; it's rarely used to monitor a common file.

SEE ALSO **DPSAddPort, DPSAddTimedEntry, DPSRemoveFD**

DPSAddNotifyPortProc

SUMMARY Set the handler function for the notify port

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSAddNotifyPortProc(**DPSPortProc *handler*, void **userData***)*

DESCRIPTION **DPSAddNotifyPortProc** registers *handler* as the function that's called when a message arrives on the notify port, the unique port, created through the **task_notify** Mach function, on which notifications (such as port death) are sent. You don't need to create the notify port yourself; **DPSAddNotifyPortProc** creates it for you if it doesn't already exist.

DPSPortProc, *handler*'s defined type, takes the form

```
void *handler(msg_header_t *msg, void *userData)
```

where *msg* is a pointer to the message that was received at the port and *userData* is the same pointer that was passed as the second argument to **DPSAddNotifyPortProc**. The *userData* pointer is provided as a convenience, allowing you to pass arbitrary data to *handler*.

The notify port can have only one handler at a time; adding a handler removes the current one. You can remove the port's handler without specifying a new one with the **DPSRemoveNotifyPortProc** function. The function's argument must match the notify port's current handler.

SEE ALSO **DPSAddPort, DPSAddTimedEntry, DPSRemoveNotifyPortProc**

DPSAddPort

SUMMARY Monitor a Mach port

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSAddPort(**port_t *port*, DPSPortProc *handler*, int *maxMsgSize*, void **userData*, int *priority***)**

DESCRIPTION **DPSAddPort** registers the function *handler* to be called each time your application asks for an event or peeks at the event queue. The function is called provided the following are true:

- The Mach port *port* must be valid and it must hold a message waiting to be read.
- *priority*, an integer from 0 to 30, must be equal to or greater than the application's current priority threshold. See **DPSAddTimedEntry** for a further explanation.

DPSPortProc, *handler*'s defined type, takes the form

```
void *handler(msg_header_t *msg, void *userData)
```

where *msg* is a pointer to the message that was received at the port and *userData* is the same pointer that was passed as the fourth argument to **DPSAddPort**. The *userData* pointer is provided as a convenience, allowing you to pass arbitrary data to *handler*.

If, within *handler*, you want to call **msg_receive** to receive further messages at the port, you must first call **DPSRemovePort** to remove the port from the system's port set. (This is because your application can't receive messages from a port that's in a port set.) After your application is finished receiving messages directly from the port, it can call **DPSAddPort** to have the system continue to monitor the port.

The contents of the message buffer *msg*, as received by *handler*, are invalid after the function returns. If you need to save any of the information that you find.

The *maxMsgSize* argument is an integer that gives the size, in bytes, of the largest message you expect to receive.

SEE ALSO **DPSAddFD, DPSAddTimedEntry, DPSRemovePort**

DPSAddTimedEntry

SUMMARY Create a timed entry

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS DPSTimedEntry **DPSAddTimedEntry(**double *period*, DPSTimedEntryProc *handler*, void **userData*, int *priority***)**

DESCRIPTION **DPSAddTimedEntry** registers *handler* as a "timed entry," a function that's called repeatedly at a given time interval. *period* determines the number of seconds between calls to the timed entry. Whenever an application based on the Application Kit attempts to retrieve events from the event

queue, it also checks (depending on *priority*) to determine whether any timed entries are due to be called. *userData* is a pointer that you can use to pass some data to the timed entry.

The function registered as *handler* has the form:

```
void *handler(DPSTimedEntry tag, double now, void *userData)
```

where *tag* is the timed entry identifier returned by **DPSAddTimedEntry**, *now* is the number of seconds since some arbitrary point in the past, and *userData* is the pointer **DPSAddTimedEntry** received when this timed entry was installed.

An application's priority threshold can be set explicitly as an integer from 0 to 31 through a call to **DPSGetEvent** or **DPSPeekEvent**. It's against this threshold that *priority* is measured (note that *priority* can be no greater than 30—the additional threshold level, 31, is provided to disallow all inter-event function calls). However, if you're using the Application Kit, you should access the event queue through Application class methods such as **getNextEvent:**. Although some of these methods let you set the priority threshold explicitly, you typically invoke the methods that set it automatically. Such methods use only three priority levels:

Constant	Meaning
NX_BASETHRESHOLD	Normal execution
NX_RUNMODALTHRESHOLD	An attention panel is being run
NX_MODALRESPTHRESHOLD	A modal event loop is being run

When applicable, you should use these constants as the value for *priority*. For example, if you want *handler* to be called during normal execution, but not if an attention panel or a modal loop is running, then you would set *priority* to NX_BASETHRESHOLD.

RETURN **DPSAddTimedEntry** returns a number identifying the timed entry or –1 to indicate an error.

SEE ALSO **DPSRemoveTimedEntry**

DPSAsynchronousWaitContext

SUMMARY Proceed asynchronously while PostScript code is executed

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSAsynchronousWaitContext(**DPSContext *context*, DPSPingProc *handler*, void **userData***userData***)**

DESCRIPTION This function is similar to the more familiar **DPSWaitContext** functions, except that rather than wait for all PostScript code to execute, it returns immediately, allowing your application to proceed while the PostScript code is executed in the background. The DPSPingProc function *handler* is called (with *context* and *userData* as its two arguments) when all the PostScript code has been executed. The DPSPingProc function takes the form

```
void *handler(DPSContext context, void *userData);
```

Warning: Be careful when you use this function; you must not send more PostScript code while waiting for the handler to be called. In general, it's best to not make any demands on the Application Kit or the Client Library if you're waiting for an asynchronous handler to return.

DPSCreateContext

SUMMARY Create a PostScript execution context

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS DPSContext **DPSCreateContext(**const char **hostName*, const char **serverName*, DPSTextProc *textProc*, DPSErrorProc *errorProc***)**

DESCRIPTION **DPSCreateContext** establishes a connection with the Window Server and creates a PostScript execution context in it. The new context becomes the current context. The first argument, *hostName*, identifies the machine that's running the Window Server; the second argument, *serverName*, identifies the Window Server that's running on that machine. With these two arguments and the help of the Mach network server **nmserver**, the Mach port for the Window Server can be identified. If *hostName* is NULL, the network server on the local machine is queried

for the Window Server's port. If *serverName* is NULL, a default name for the Window Server is used.

The last two arguments, *textProc* and *errorProc*, refer to call-back functions (defined in the Client Library specification) that handle text returned from the Window Server and errors generated on either side of the connection.

For an application that's based on the Application Kit, you could create an additional context by making this call:

```
DPSContext c;

c = DPSCreateContext(NXGetDefaultValue([NXApp appName], "NXHost"),
                        XGetDefaultValue([NXApp appName], "NXPSName"),
                        NULL,
                        NULL);
```

This example queries the application's default values for the identity of the host machine and the Window Server. By doing this, the new context is created in the correct Window Server even if that Server is not on the same machine as the application process.

The context that **DPSCreateContext** creates allocates memory from the default allocation zone. Also, when there's difficulty creating the context, **DPSCreateContext** waits up to 60 seconds before raising an exception. If you want to change either of these parameters, use **DPSCreateContextWithTimeoutFromZone**. Its two additional arguments let you specify the zone for the context to use when allocating context-specific data and a timeout value in milliseconds.

This function returns the newly created DPSContext structure.

EXCEPTIONS

DPSCreateContext raises a dps_err_outOfMemory exception if it encounters difficulty allocating ports or other resources for the new context. It raises a dps_err_cantConnect exception if it can't return a context within the timeout period.

SEE ALSO

DPSCreateContextWithTimeoutFromZone, DPSCreateNonsecureContext, DPSCreateStreamContext

DPSCreateContextWithTimeoutFromZone

SUMMARY Create a PostScript execution context

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS DPSContext **DPSCreateContextWithTimeoutFromZone(**const char **hostName*, const char **serverName*, DPSTextProc *textProc*, DPSErrorProc *errorProc*, int *timeout*, NSZone **zone***)*

DESCRIPTION **DPSCreateContextWithTimeoutFromZone** is identical to DPSCreateContext except that it accepts two additional arguments that let you specify the zone to use when allocating context-specific data and a timeout value other than the default value of 60 seconds. Specify the new *timeout* value in milliseconds,

This function returns the newly created DPSContext structure.

EXCEPTIONS **DPSCreateContextWithTimeoutFromZone** raises a dps_err_outOfMemory exception if it encounters difficulty allocating ports or other resources for the new context. It raises a dps_err_cantConnect exception if it can't return a context within the timeout period.

SEE ALSO **DPSCreateContext, DPSCreateNonsecureContext, DPSCreateStreamContext**

DPSCreateNonsecureContext

SUMMARY Create a PostScript execution context

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS DPSContext **DPSCreateNonsecureContext(**const char **hostName*, const char **serverName*, DPSTextProc *textProc*, DPSErrorProc *errorProc*, int *timeout*, NSZone **zone***)*

DESCRIPTION **DPSCreateNonsecureContext** creates a "nonsecure" context in which you can use PostScript operators that are normally disallowed. The most significant of these are operators that let you write files.

Few programmers will need to call this function directly: The Application Kit manages contexts for programs based on the Kit. For example, when an application is launched, its Application object calls **DPSCreateContext** to create a context in the Window Server. Similarly, to print a View the Kit calls **DPSCreateStreamContext** to temporarily redirect PostScript code from the View to a stream.

This function returns the newly created DPSContext structure.

SEE ALSO **DPSCreateContext, DPSCreateContextWithTimeoutFromZone, DPSCreateStreamContext**

DPSCreateStreamContext

SUMMARY Create a PostScript execution context

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS DPSContext **DPSCreateStreamContext(**NXStream **stream*, int *debugging*, DPSProgramEncoding *progEnc*, DPSNameEncoding *nameEnc*, DPSErrorProc *errorProc***)**

DESCRIPTION **DPSCreateStreamContext** is similar to **DPSCreateContext**, except that the new context is actually a connection from the client application to a stream. This connection becomes the current context. PostScript code that the application generates is sent to the stream rather than to the Window Server. The first argument, *stream*, is a pointer to an NXStream structure, as created by **NXOpenFile** or **NXMapFile**. The *debugging* argument is intended for debugging purposes but is not currently implemented. *progEnc* and *nameEnc* specify the type of program and user-name encodings to be used for output to the stream. The last argument, *errorProc*, identifies the procedure that's called when errors are generated.

Few programmers will need to call this function directly: The Application Kit manages contexts for programs based on the Kit. For example, when an application is launched, its Application object calls **DPSCreateContext** to create a context in the Window Server. Similarly, to print a View the Kit calls **DPSCreateStreamContext** to temporarily redirect PostScript code from the View to a stream.

This function returns the newly created DPSContext structure.

SEE ALSO **DPSCreateContext, DPSCreateContextWithTimeoutFromZone, DPSCreateNonsecureContext**

DPSDefineUserObject

SUMMARY Create a user object

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS int **DPSDefineUserObject(**int *index***)**

DESCRIPTION **DPSDefineUserObject** associates *index* with the PostScript object that's on the top of the operand stack, thereby creating a user object (as defined by the PostScript language). If *index* is 0, the object is assigned the next available index number. The function returns the new index, which can then be passed to a **pswrap**-generated function that takes a user object.

Warning: To avoid coming into conflict with user objects defined by the Client Library or Application Kit, use **DPSDefineUserObject** rather than the PostScript operator **defineuserobject** or the single-operator functions **DPSdefineuserobject** and **PSdefineuserobject**.

RETURN **DPSDefineUserObject**, if successful in assigning an index, returns the index that the object was assigned. If unsuccessful, it returns 0.

SEE ALSO **DPSUndefineUserObject**

DPSDiscardEvents

SUMMARY Discard events from the Window Server

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSDiscardEvents(**DPSContext *context*, int *mask***)**

DESCRIPTION **DPSDiscardEvents**'s two parameters, *context* and *mask*, are the same as those for **DPSGetEvent** and **DPSPeekEvent**. **DPSDiscardEvents** removes from the application's event queue those records whose event types match *mask* and whose context matches *context*.

RETURN **DPSGetEvent** and **DPSPeekEvent** return 1 if they are successful in accessing an event record and 0 if they aren't.

SEE ALSO **DPSGetEvent, DPSPeekEvent, DPSAddFD, DPSAddPort, DPSAddTimedEntry, DPSPostEvent**

DPSDoUserPath

SUMMARY Send an encoded PostScript path to the Window Server

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSDoUserPath(**void **coords*, int *numCoords*, DPSNumberFormat *numType*, unsigned char **ops*, int *numOps*, void **bbox*, int *action***)**

DESCRIPTION **DPSDoUserPath** sends an encoded user path to the Window Server and then executes, upon that path, the operator specified by *action*. The use of this function, rather than the analogous step-by-step path construction, is encouraged; rendering an encoded path is much more efficient than executing the individual PostScript operators that would otherwise be needed.

An encoded user path consists of an array of coordinate values, a sequence of PostScript operators, and a bounding box specification. The values in the coordinate array are used as operands to the operators; the operands are distributed to the operators in the order that they're given. The resulting path must fit within the bounding box.

The coordinates, operators, and bounding box are given by the function's first five arguments:

- The array of coordinate values is given by *coords*.
- *numCoords* is the number of elements in *coords*.
- *numType* specifies the data type of the coordinates, as described below. All the values in *coords* must be of the same type.
- *ops* is the sequence of PostScript operators, represented by constants as listed below.
- The bounding box is defined by the four coordinate values that you pass as an array in the *bbox* argument. These are passed as operands to the **setbbox** operator. (If you don't supply a **setbbox** as part of the *ops* sequence, one is inserted for you.)

The following integer constants represent the data types that you can pass as the *numType* argument:

Constant	Meaning
dps_float	single-precision floating-point number
dps_long	32-bit integer
dps_short	8-bit integer

You can also specify 16- and 32-bit fixed-point real numbers. For 16-bit fixed-point numbers, use **dps_short** plus the number of bits in the fractional portion. For 32-bit fixed-point numbers, use **dps_long** plus the number of bits in the fractional portion.

These constants are provided for *ops*:

dps_setbbox
dps_moveto
dps_rmoveto
dps_lineto
dps_rlineto
dps_curveto
dps_rcurveto
dps_arc
dps_arcn
dps_arct
dps_closepath
dps_ucache

Once the user path has been constructed, the operator specified by *action* is executed. The value of *action* is an index into Display PostScript's encoded system names; the following constants, provided as a convenience, represent the most commonly used actions:

dps_uappend
dps_ufill
dps_ueofill
dps_ustroke
dps_ustrokepath
dps_inufill
dps_inueofill
dps_inustroke
dps_def
dps_put

The following program fragment demonstrates the use of **DPSDoUserPath** as it creates and strokes a user path (an isosceles triangle) within a bounding rectangle whose lower left corner is located at (0, 0) and whose width and height are 200.

```
short  coords[6] = {0, 0, 200, 0, 100, 200};
char  ops[4] = {dps_moveto, dps_lineto,dps_lineto, dps_closepath};
short  bbox[4] = {0, 0, 200, 200};

DPSDoUserPath(coords, 6, dps_short, ops, 4, bbox, dps_ustroke);
```

Note: If an application calls **DPSDoUserPath** with large values (~10,000-20,000) of *numCoords* and/or *numOps*, it may generate a Display PostScript error.

SEE ALSO **DPSDoUserPathWithMatrix**

DPSDoUserPathWithMatrix

SUMMARY Send an encoded PostScript path to the Window Server

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSDoUserPathWithMatrix(**void **coords*, int *numCoords*, DPSNumberFormat *numType*, unsigned char **ops*, int *numOps*, void **bbox*, int *action*, float *matrix*[6]**)**

DESCRIPTION **DPSDoUserPathWithMatrix** sends an encoded user path to the Window Server and then executes, upon that path, the operator specified by *action*. The use of this function, rather than the analogous step-by-step path construction, is encouraged; rendering an encoded path is much more efficient than executing the individual PostScript operators that would otherwise be needed.

An encoded user path consists of an array of coordinate values, a sequence of PostScript operators, and a bounding box specification. The values in the coordinate array are used as operands to the operators; the operands are distributed to the operators in the order that they're given. The resulting path must fit within the bounding box.

The coordinates, operators, and bounding box are given by the function's first five arguments:

- The array of coordinate values is given by *coords*.
- *numCoords* is the number of elements in *coords*.

- *numType* specifies the data type of the coordinates, as described below. All the values in *coords* must be of the same type.
- *ops* is the sequence of PostScript operators, represented by constants as listed below.
- The bounding box is defined by the four coordinate values that you pass as an array in the *bbox* argument. These are passed as operands to the **setbbox** operator. (If you don't supply a **setbbox** as part of the *ops* sequence, one is inserted for you.)

The following integer constants represent the data types that you can pass as the *numType* argument:

Constant	Meaning
dps_float	single-precision floating-point number
dps_long	32-bit integer
dps_short	8-bit integer

You can also specify 16- and 32-bit fixed-point real numbers. For 16-bit fixed-point numbers, use **dps_short** plus the number of bits in the fractional portion. For 32-bit fixed-point numbers, use **dps_long** plus the number of bits in the fractional portion.

These constants are provided for *ops*:

dps_setbbox
dps_moveto
dps_rmoveto
dps_lineto
dps_rlineto
dps_curveto
dps_rcurveto
dps_arc
dps_arcn
dps_arct
dps_closepath
dps_ucache

Once the user path has been constructed, the operator specified by *action* is executed. The value of *action* is an index into Display PostScript's encoded system names; the following constants, provided as a convenience, represent the most commonly used actions:

dps_uappend
dps_ufill
dps_ueofill
dps_ustroke
dps_ustrokepath
dps_inufill
dps_inueofill
dps_inustroke
dps_def
dps_put

DPSDoUserPathWithMatrix's *matrix* argument represents the transformation matrix operand used by the **ustroke**, **inustroke**, and **ustrokepath** operators. If *matrix* is NULL, the argument is ignored.

DPSFlush

SUMMARY Send PostScript data to the Window Server

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSFlush**

DESCRIPTION **DPSFlush** flushes the application's output buffer, forcing any buffered PostScript code or data to the Window Server.

SEE ALSO **DPSSendEOF**

DPSGetEvent

SUMMARY Access events from the Window Server

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS int **DPSGetEvent(**DPSContext *context*, NXEvent **anEvent*, int *mask*, double *timeout*, int *threshold***)**

DESCRIPTION **DPSGetEvent** and **DPSPeekEvent** are macros that access event records in an application's event queue. These routines are provided primarily for programs that don't use the Application Kit. An application based on the Kit should use the corresponding Application class methods (such as **getNextEvent:** and **peekNextEvent:into:**) so that it can be journaled. **DPSDiscardEvents** removes all event records of a specified type from the queue.

DPSGetEvent and **DPSPeekEvent** differ only in how they treat the accessed event record. **DPSGetEvent** removes the record from the queue after making its data available to the application; **DPSPeekEvent** leaves the record in the queue.

DPSGetEvent and **DPSPeekEvent** take the same parameters. The first, *context*, represents a PostScript execution context within the Window Server. Virtually all applications have only one execution context, which can be returned using **DPSGetCurrentContext**. Applications having more than one execution context can use the constant DPS_ALLCONTEXTS to access events from all contexts belonging to them.

The second argument, *anEvent*, is a pointer to an event record. If **DPSGetEvent** or **DPSPeekEvent** is successful in accessing an event record, the record's data is copied into the storage referred to by *anEvent*.

mask determines the types of events sought. See the section "Types and Constants" for a list of the constants that represent the event type masks. To check for more than one type of event, you combine individual constants using the bitwise OR operator.

If an event matching the event mask isn't available in the queue, **DPSGetEvent** or **DPSPeekEvent** waits until one arrives or until *timeout* seconds have elapsed, whichever occurs first. The value of *timeout* can be in the range of 0.0 to NX_FOREVER. If *timeout* is 0.0, the routine returns an event only if one is waiting in the queue when the routine asks for it. If *timeout* is NX_ FOREVER, the routine waits until an appropriate event arrives before returning.

The last argument, *threshold*, is an integer in the range 0 through 31 that determines which other services may be provided during a call to **DPSGetEvent** or **DPSPeekEvent**.

Requests for services are registered by the functions **DPSAddTimedEntry**, **DPSAddPort**, and **DPSAddFD**. Each of these functions takes an argument specifying a priority level. If this level is equal to or greater than *threshold*, the service is provided before **DPSGetEvent** or **DPSPeekEvent** returns.

RETURN **DPSGetEvent** returns 1 if it is successful in accessing an event record and 0 if it isn't.

SEE ALSO **DPSPeekEvent, DPSDiscardEvents, DPSAddFD, DPSAddPort, DPSAddTimedEntry, DPSPostEvent**

DPSInterruptContext

Warning: This function is unimplemented in the NEXTSTEP version of the Client Library.

DPSNameFromTypeAndIndex

SUMMARY Access the system and user name tables

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS const char ***DPSNameFromTypeAndIndex(**short *type*, int *index***)**

DESCRIPTION **DPSNameFromTypeAndIndex** returns the text associated with *index* from the system or user name table. If *type* is –1, the text is returned from the system name table; if *type* is 0, it's returned from the user name table.

The name tables are used primarily by the Client Library and **pswrap**; few programmers will access them directly.

RETURN This function returns a read-only character string.

DPSPeekEvent

SUMMARY Access events from the Window Server

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS int **DPSPeekEvent(**DPSContext *context*, NXEvent **anEvent*, int *mask*, double *timeout*, int *threshold***)**

DESCRIPTION **DPSGetEvent** and **DPSPeekEvent** are macros that access event records in an application's event queue. These routines are provided primarily for programs that don't use the Application Kit. An application based on the Kit should use the corresponding Application class methods (such as **getNextEvent:** and **peekNextEvent:into:**) so that it can be journaled. **DPSDiscardEvents** removes all event records of a specified type from the queue.

DPSGetEvent and **DPSPeekEvent** differ only in how they treat the accessed event record. **DPSGetEvent** removes the record from the queue after making its data available to the application; **DPSPeekEvent** leaves the record in the queue.

DPSGetEvent and **DPSPeekEvent** take the same parameters. The first, *context*, represents a PostScript execution context within the Window Server. Virtually all applications have only one execution context, which can be returned using **DPSGetCurrentContext**. Applications having more than one execution context can use the constant DPS_ALLCONTEXTS to access events from all contexts belonging to them.

The second argument, *anEvent*, is a pointer to an event record. If **DPSGetEvent** or **DPSPeekEvent** is successful in accessing an event record, the record's data is copied into the storage referred to by *anEvent*.

mask determines the types of events sought. See the section "Types and Constants" for a list of the constants that represent the event type masks. To check for more than one type of event, you combine individual constants using the bitwise OR operator.

If an event matching the event mask isn't available in the queue, **DPSGetEvent** or **DPSPeekEvent** waits until one arrives or until *timeout* seconds have elapsed, whichever occurs first. The value of *timeout* can be in the range of 0.0 to NX_FOREVER. If *timeout* is 0.0, the routine returns an event only if one is waiting in the queue when the routine asks for it. If *timeout* is NX_ FOREVER, the routine waits until an appropriate event arrives before returning.

The last argument, *threshold*, is an integer in the range 0 through 31 that determines which other services may be provided during a call to **DPSGetEvent** or **DPSPeekEvent**.

Requests for services are registered by the functions **DPSAddTimedEntry**, **DPSAddPort**, and **DPSAddFD**. Each of these functions takes an argument specifying a priority level. If this level is equal to or greater than *threshold*, the service is provided before **DPSGetEvent** or **DPSPeekEvent** returns.

RETURN **DPSPeekEvent** returns 1 if it is successful in accessing an event record and 0 if it isn't.

SEE ALSO **DPSGetEvent, DPSDiscardEvents, DPSAddFD, DPSAddPort, DPSAddTimedEntry, DPSPostEvent**

DPSPostEvent

SUMMARY Create an event

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS int **DPSPostEvent(**NXEvent **anEvent*, int *atStart***)**

DESCRIPTION **DPSPostEvent** lets you add an event record to your application's event queue without involving the Window Server. *anEvent* is a pointer to the event record to be added. *atStart* specifies where the new record will be placed in relation to any other records in the queue. If *atStart* is TRUE, the event is posted in front of all others and so will be the next one your application receives. If *atStart* is FALSE, the event is posted behind all others and so won't be returned until events that precede it are processed.

You can free, reuse, or otherwise mangle *anEvent* after you've posted it without fear of corrupting the posted record. **DPSEvent** copies the record it receives and posts the copy.

Note that event records you post using **DPSPostEvent** aren't filtered by an event filter function set with **DPSSetEventFunc**.

RETURN **DPSPostEvent** returns 0 if successful in posting the event record; it returns –1 if unsuccessful in posting the record because the event queue is full.

SEE ALSO **DPSSetEventFunc**

DPSPrintError

SUMMARY Print error messages

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSPrintError(**FILE **fp*, const DPSBinObjSeq *error***)**

DESCRIPTION **DPSPrintError** and **DPSPrintErrorToStream** format and print error messages received from a PostScript execution context in the Window Server. The error message is extracted from the binary object sequence *error*. **DPSPrintError** prints the error message to the file identified by *fp*; **DPSPrintErrorToStream** prints the error message to *stream*.

You rarely need to call this function directly. However, if you reset the error handler for a PostScript execution context, the new handler you install could use this function to process errors that it receives.

SEE ALSO **DPSPrintErrorToStream**

DPSPrintErrorToStream

SUMMARY Print error messages

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSPrintErrorToStream(**NXStream **stream*, const DPSBinObjSeq *error***)**

DESCRIPTION **DPSPrintError** and **DPSPrintErrorToStream** format and print error messages received from a PostScript execution context in the Window Server. The error message is extracted from the binary object sequence *error*. **DPSPrintError** prints the error message to the file identified by *fp*; **DPSPrintErrorToStream** prints the error message to *stream*.

You rarely need to call this function directly. However, if you reset the error handler for a PostScript execution context, the new handler you install could use this function to process errors that it receives.

SEE ALSO **DPSPrintError**

DPSRemoveFD

SUMMARY Stop monitoring a file descriptor

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSRemoveFD(**int *fd***)**

DESCRIPTION **DPSRemoveFD** removes the specified file descriptor from the list of those that the application will check.

SEE ALSO **DPSAddFD**

DPSRemoveNotifyPortProc

SUMMARY Removes the handler function for the notify port

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSRemoveNotifyPortProc(**DPSPortProc *handler***)**

DESCRIPTION Removes the notify port's handler without specifying a new one. The *handler* argument must match the notify port's current handler.

SEE ALSO **DPSAddNotifyPortProc**

DPSRemovePort

SUMMARY Remove the Mach port being monitored.

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSRemovePort(**port_t *port***)**

DESCRIPTION **DPSRemovePort** removes the specified Mach port from the list of those that the application will check.

SEE ALSO **DPSAddPort**

DPSRemoveTimedEntry

SUMMARY Create a timed entry

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSRemoveTimedEntry(**DPSTimedEntry *tag***)**

DESCRIPTION **DPSRemoveTimedEntry** removes a previously registered timed entry.

SEE ALSO **DPSAddTimedEntry**

DPSResetContext

Warning: This function is unimplemented in the NEXTSTEP version of the Client Library.

DPSSendEOF

SUMMARY Send PostScript data to the Window Server

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSSendEOF(**DPSContext *context***)**

DESCRIPTION **DPSSendEOF** sends a PostScript end-of-file marker to the given context. The connection to the context isn't closed or disturbed in any way by this function.

SEE ALSO **DPSFlush**

DPSSetDeadKeysEnabled

SUMMARY Allow dead key processing for a context's events

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSSetDeadKeysEnabled(**DPSContext *context*, int *flag***)**

DESCRIPTION **DPSSetDeadKeysEnabled** turns dead key processing on or off for *context*. If flag is 0, dead key processing is turned off; otherwise, it's turned on (the default).

Dead key processing is a technique for extending the range of characters that can be entered from the keyboard. In NEXTSTEP, it provides one way for users to enter accented characters. For example, a user can type Alternate-e followed by the letter "e" to produce the letter "é". The first keyboard input, Alternate-e, seems to have no effect—it's the "dead key". However, it signals Client Library routines that it and the following character should be analyzed as a pair. If, within NEXTSTEP, the pair of characters has been associated with a third character, a keyboard event record representing the third character is placed in the application's event queue, and the first two event records are discarded. If there is no such association between the two characters, the two event records are added to the event queue.

DPSSetEventFunc

SUMMARY Set the function that filters events

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS DPSEventFilterFunc **DPSSetEventFunc(**DPSContext *context*, DPSEventFilterFunc *func***)**

DESCRIPTION **DPSSetEventFunc** establishes the function *func* as the function to be called when an event record is returned from the PostScript context *context* in the Window Server. The registered function is called before the event record is put in the event queue. If the registered function returns 0, the record is discarded. If the registered function returns 1, the record is passed on for further processing.

Only event records coming from the Window Server are filtered by the registered function. Records that you post to the event queue using **DPSPostEvent** aren't affected.

A DPSEventFilterFunc function takes the following form:

```
int *func(NXEvent *anEvent)
```

RETURN **DPSSetEventFunc** returns a pointer to the previously registered event function. This lets you chain together the current and previous event functions.

SEE ALSO **DPSPostEvent**

DPSSetTracking

SUMMARY Coalesce mouse events

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS int **DPSSetTracking(**int *flag***)**

DESCRIPTION **DPSSetTracking** turns mouse event-coalescing on or off for the current context. If *flag* is 0, coalescing is turned off; otherwise, it's turned on (the default).

Event coalescing is an optimization that's useful when tracking the mouse. When the mouse is moved, numerous events flow into the event queue. To reduce the number of events awaiting removal by the application, adjacent mouse-moved events are replaced by the most recent event of the contiguous group. The same is done for left and right mouse-dragged events, with the addition that a mouse-up event replaces mouse-dragged events that come before it in the queue.

RETURN **DPSSetTracking** returns the previous state of the event-coalescing switch.

DPSStartWaitCursorTimer

SUMMARY Initiate a count down for the wait cursor

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSStartWaitCursorTimer**

DESCRIPTION **DPSStartWaitCursorTimer** triggers the mechanism that displays a wait cursor when an application is busy and can't respond to user input. In most cases, wait cursor support is automatic: You'll only need to call this function if your application starts a time-consuming operation that's not initiated by a user-generated event.

Client Library routines and the Window Server cooperate to display the wait cursor whenever more than a preset amount of time elapses between the time an application takes an event record from the event queue and the time the application is again ready to consume events. However, when an application starts an operation that isn't initiated by an event—such as one caused by receiving a Mach message or by processing data from a file (see **DPSAddPort** and **DPSAddFD**)—the wait cursor mechanism is bypassed. To ensure proper wait cursor behavior in these cases, call **DPSStartWaitCursorTimer** before beginning the time-consuming operation.

SEE ALSO **DPSAddFD, DPSAddPort**

DPSSynchronizeContext

SUMMARY Synchronize a context with your application

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS int **DPSSynchronizeContext(**DPSContext *context*, int *flag***)**

DESCRIPTION **DPSSynchronizeContext** causes **DPSWaitContext** to be called after each **pswrap** function is called, thus synchronizing the PostScript context with your application.

DPSTraceContext

SUMMARY Trace data and events

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS int **DPSTraceContext(**DPSContext *context*, int *flag***)**

DESCRIPTION **DPSTraceContext** controls the tracing of data between a PostScript execution context (or contexts) in the Window Server and an application process.

The first argument, *context*, specifies the context to be traced. An application's single context can be returned with **DPSGetCurrentContext**. Applications having more than one execution context can use the constant DPS_ALLCONTEXTS to trace all contexts belonging to them.

The second argument, *flag*, determines whether tracing is enabled. When data tracing is enabled, a copy of the PostScript code generated by an application and the values returned to it by the Window Server is sent to standard error. Values returned to the application are marked by the prepended string:

```
% value returned ==>
```

For applications based on the Application Kit, there are two preferable methods for turning on data tracing: You can use the NXShowPS command-line switch when you launch an application from Terminal. Alternatively, when you run the application under GDB, you can use the **showps** and **shownops** commands to control tracing output.

Only one tracing context can be created for the supplied *context*. If you attempt to create additional tracing contexts for a context that's already being traced, no new context is created and **DPSTraceContext** returns –1.

RETURN **DPSTraceContext** returns 0 if successful in creating a tracing context, or –1 if not.

SEE ALSO **DPSTraceEvents**

DPSTraceEvents

SUMMARY Trace data and events

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSTraceEvents(**DPSContext *context*, int *flag***)**

DESCRIPTION **DPSTraceEvents** controls the tracing of events between a PostScript execution context (or contexts) in the Window Server and an application process.

The first argument, *context*, specifies the context to be traced. An application's single context can be returned with **DPSGetCurrentContext**. Applications having more than one execution context can use the constant DPS_ALLCONTEXTS to trace all contexts belonging to them.

The second argument, *flag*, determines whether tracing is enabled. When event tracing is enabled, information about each event that the application receives is sent to standard error. For example, for a left mouse-down event the listing might look like this:

```
Receiving: LMouseDown at: 343.0,69.0 time: 1271899
   flags: 0x0 win: 6 ctxt: 76128 data: 1111,1
```

The listing displays the fields of the event record: type, location, time, flags, local window number, PostScript execution context, and data. The contents of the data field listing depends on the event type; for instance, in the preceding example the event number and the click count were displayed.

To enable event tracing, you can use the **NXTraceEvents** command-line switch when you launch an application from Terminal. Alternatively, when you run the application under GDB, you can use the **traceevents** and **tracenoevents** commands to control event-tracing output.

SEE ALSO **DPSTraceContext**

DPSUndefineUserObject

SUMMARY Remove a user object

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSUndefineUserObject(**int *index***)**

DESCRIPTION **DPSUndefineUserObject** removes the association between *index* and the PostScript object it refers to, thus destroying the user object. By destroying a user object that's no longer needed, you can let the garbage collector reclaim the previously associated PostScript object.

SEE ALSO **DPSDefineUserObject**

NX_EVENTCODEMASK

SUMMARY Generate an event mask for an event type

DECLARED IN dpsclient/event.h

SYNOPSIS int **NX_EVENTCODEMASK(**int *type***)**

DESCRIPTION This handy utility macro returns an event mask that corresponds to the given (single) event type.

Single-Operator Functions

Note: This section has not been updated and has not received recent technical review. It is included in this release to test the linkage between application development tools and on-line documentation. The information in this section should be considered at best preliminary and subject to change. An updated version of this file will be included in the next release.

PSadjustcursor(float *dx*, float *dy***)**
PSalphaimage(void**)**
PSbasetocurrent(float *bx*, float *by*, float **cx*, float **cy***)**
PSbasetoscreen(float *bx*, float *by*, float **sx*, float **sy***)**
PSbuttondown(boolean **isdown***)**
PScleartrackingrect(int *trectnum*, userobject *gstate***)**
PScomposite(float *srcx*, float *srcy*, float *width*, float *height*, userobject *srcgstate*, float *destx*, float *desty*, int *op***)**

The value passed as *op* should be one of the following:

NX_CLEAR	NX_SIN	NX_SATOP
NX_COPY	NX_DIN	NX_DATOP
NX_SOVER	NX_SOUT	NX_PLUSD
NX_DOVER	NX_DOUT	NX_PLUSL
NX_XOR		

PScompositerect(float *destx*, float *desty*, float *width*, float *height*, int *op***)**

The value passed as *op* should be one *of the constants listed under* **PScomposite**, plus NX_HIGHLIGHT.

PScountframebuffers(int **count***)**
PScountscreenlist(int *context*, int **count***)**
PScountwindowlist(int *context*, int **count***)**
PScurrentactiveapp(int **context***)**
Warning: Don't use this function if you're using the Application Kit.

PScurrentalpha(float **coverage***)**
PScurrentdefaultdepthlimit(int **depth***)**

Warning: Don't use this function if you're using the Application Kit.

PScurrentdeviceinfo(userobject *window*, int **min*, int **max*, boolean **iscolor***)*
PScurrenteventmask(userobject *window*, int **mask***)*
Warning: Don't use this function if you're using the Application Kit.

PScurrentframebuffertransfer(void**)**
PScurrentmouse(userobject *window*, float **x*, float **y***)*
Warning: Don't use this function if you're using the Application Kit.

PScurrentowner(userobject *window*, int **context***)*
PScurrentshowpageprocedure(void)
PScurrentrusage(float **ctime*, float **utime*, float **stime*, int **msgsend*, int **msgrcv*, int **nsignals*, int **nvcsw*, int **nivcsw***)*
PScurrenttobase(float *cx*, float *cy*, float **bx*, float **by***)*
PScurrenttoscreen(float *cx*, float *cx*, float **sx*, float **sy***)*
PScurrentuser(int **uiId*, int **gid***)*
PScurrentwaitcursorenabled(boolean **isenabled***)*
PScurrentwindow(userobject **window***)*
PScurrentwindowalpha(userobject *window*, int **alpha***)*
PScurrentwindowbounds(userobject *window*, float **x*, float **y*, float **width*, float **height***)*
Warning: Don't use this function if you're using the Application Kit.

PScurrentwindowdepth(userobject *window*, int **depth***)*
Warning: Don't use this function if you're using the Application Kit.

PScurrentwindowdepthlimit(userobject *window*, int **depth***)*
Warning: Don't use this function if you're using the Application Kit.

PScurrentwindowdict(userobject *window***)**
Warning: Don't use this function if you're using the Application Kit.

PScurrentwindowlevel(userobject *window*, int **level***)*
PScurrentwriteblock(bool **doesblock***)*
PSdissolve(float *srcx*, float *srcy*, float *sourceWidth*, float *width*, userobject *srcgstate*, float *destx*, float *desty*, float *delta***)**
PSdumpwindow(int *dumplevel*, userobject *window***)**
Warning: Don't use this function if you're using the Application Kit.

PSdumpwindows(int *dumplevel*, userobject *context***)**

Warning: Don't use this function if you're using the Application Kit.

PSfindwindow(float *x*, float *y*, int *place*, userobject *otherwindow*, float **x*', float **y*', userobject **window*, boolean **found***)*

The value passed as place should be one of the following:

NX_ABOVE
NX_BELOW

PSflushgraphics(void**)**
Warning: Don't use this function if you're using the Application Kit.

PSframebuffer(int *index*, int *stringlen*, char *string*[], int **slot*, int **unit*, int **romid*, int **x*, int **y*, int **width*, int **height*, int **maxdepth***)*
PSfrontwindow(int **window***)*
Warning: Don't use this function if you're using the Application Kit.

PShidecursor(void**)**
PShideinstance(float *x*, float *y*, float *width*, float *height***)**
PSmachportdevice(int *width*, int *height*, const int *bbox*[], int *bboxSize*, const float *matrix*[], const char **hostname*, const char **portname*, const char **pixelencoding***)*
PSmovewindow(float *x*, float *y*, userobject *window***)**
Warning: Don't use this function if you're using the Application Kit.

PSnewinstance(void**)**
PSnextrelease(int *size*, char *string*[]**)**
PSobscurecursor(void**)**
PSorderwindow(int *place*, userobject *otherwindow*, int *window***)**
Warning: Don't use this function if you're using the Application Kit.

The value passed as place should be one of the following:

NX_ABOVE
NX_BELOW
NX_OUT

PSosname(int *size*, char *string*[]**)**
PSostype(int **type***)*
PSplacewindow(float *x*, float *y*, float *width*, float *height*, userobject *window***)**†

Warning: Don't use this function if you're using the Application Kit.

PSplaysound(const char **soundname*, int *priority***)**
PSposteventbycontext(int *type*, float *x*, float *y*, int *time*, int *flags*, int *window*, int *subtype*, int *misc0*, int *misc1*, int *context*, boolean **success* **)**
PSreadimage(void**)**
PSrevealcursor(void**)**
PSrightbuttondown(int **isdown***)**
PSrightstilldown(int *eventnum*, boolean **stilldown***)**
PSscreenlist(int *context*, int *count*, int *array*[]**)**
PSscreentobase(float *sx*, float *sy*, float **bx*, float **by***)**
PSscreentocurrent(float *sx*, float *sy*, float **cx*, float **cy***)**
PSsetactiveapp(int *context***)**
Warning: Don't use this function if you're using the Application Kit.

PSsetalpha(float *coverage***)**
PSsetautofill(boolean *flag*, userobject *window***)**
PSsetcursor(float *x*, float *y*, float *mx*, float *my***)**
PSsetdefaultdepthlimit(int *depth***)**
Warning: Don't use this function if you're using the Application Kit.

PSseteventmask(int *mask*, userobject *window***)**
Warning: Don't use this function if you're using the Application Kit.

See the constants listed under "Event Type Masks" in the section "Types and Constants" for a list of *mask* values.

PSsetexposurecolor(void**)**
PSsetflushexposures(boolean *flag***)**
Warning: Don't use this function if you're using the Application Kit.

PSsetframebuffertransfer(void)
PSsetinstance(boolean *flag***)**
PSsetmouse(float *x*, float *y***)**
PSsetowner(userobject *context*, userobject *window***)**
PSsetsendexposed(boolean *flag*, userobject *window***)**†
Warning: Don't use this function if you're using the Application Kit.

PSsetshowpageprocedure(int *window***)**

Warning: Don't use this function if you're using the Application Kit.

PSsettrackingrect(float *x*, float *y*, float *width*, float *height*, boolean *leftbool*, boolean *rightbool*, boolean *insidebool*, int *userdata*, int *trectnum*, userobject *gstate***)**

Note: Only the Form 1 version of **the settrackingrect** operator is offered as a C function.

PSsetwaitcursorenabled(boolean *flag***)**
PSsetwindowdepthlimit(int *depth*, userobject *window***)**
Warning: Don't use this function if you're using the Application Kit.

PSsetwindowdict(userobject *window***)**
Warning: Don't use this function if you're using the Application Kit.

PSsetwindowlevel(int *level*, userobject *window***)**
PSsetwindowtype(int *type*, userobject *window***)**
Warning: Don't use this function if you're using the Application Kit.

PSsetwriteblock(int *flag***)**
PSshow(const char **string***)**
PSshowcursor(void**)**
PSsizeimage(float *x*, float *y*, float *width*, float *height*, int **pixelswide*, int **pixelshigh*, int **bits/sample*, float *matrix*[], boolean **multiproc*, int **ncolors***)**
PSstilldown(int *eventnum*, boolean **stilldown***)**
PStermwindow(userobject *window***)**
Warning: Don't use this function if you're using the Application Kit.

PSwindow(float *x*, float *y*, float *width*, float *height*, int *type*, int **window***)**
Warning: Don't use this function if you're using the Application Kit.

PSwindowdevice(userobject *window***)**
PSwindowdeviceround(userobject *window***)**
PSwindowlist(int *context*, int *count*, int *subarray[]***)**

PostScript Operators

Note: This section has not been updated and has not received recent technical review. It is included in this release to test the linkage between application development tools and on-line documentation. The information in this section should be considered at best preliminary and subject to change. An updated version of this file will be included in the next release.

adjustcursor

SYNOPSIS *dx dy* **adjustcursor** –

Moves the cursor location by (*dx*, *dy*) from its current location. *dx* and *dy* are given in the current coordinate system. If the current device isn't a window, the **invalidid** error is executed.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **currentmouse, setmouse**

alphaimage

SYNOPSIS *pixelswide pixelshigh bits/sample matrix datasrc0 [...datasrcn] multiproc ncolors* **alphaimage** –

Renders an image whose samples include an alpha component. (Most programmers should use **NXImageBitmap()** instead of **alphaimage**.) This operator is similar to the standard **colorimage** operator (as documented by Adobe Systems). However, note the following:

- When supplying the data components, alpha is always given last—either as the last data source (*datasrcn*) if the data is given in separate vectors, or as the last element in a set of interleaved data.
- The *ncolors* operand doesn't account for alpha—the value of *ncolors* is the number of color components only.

ERRORS **invalidid**, **limitcheck**, **rangecheck**, **stackunderflow**, **typecheck**, **undefined**, **undefinedresult**

basetocurrent

SYNOPSIS *bx by* **basetocurrent** *cx cy*

Converts (*bx*, *by*) from the current window's base coordinate system to its current coordinate system. If the current device isn't a window, the **invalidid** error is executed.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **basetoscreen, currenttobase, currenttoscreen, screentobase, screentocurrent**

basetoscreen

SYNOPSIS *bx by* **basetoscreen** *sx sy*

Converts (*bx*, *by*) from the current window's base coordinate system to the screen coordinate system. If the current device isn't a window, the **invalidid** error is executed.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **basetocurrent, currenttobase, currenttoscreen, screentobase, screentocurrent**

buttondown

SYNOPSIS – **buttondown** *isdown*

Returns *true* if the left or only mouse button is currently down; otherwise it returns *false*.

Note: To test whether the mouse button is still down from a mouse-down event, use **stilldown** instead of **buttondown**; **buttondown** will return *true* even if the mouse button has been released and pressed again since the original mouse-down event.

ERRORS none

SEE ALSO **currentmouse, rightbuttondown, rightstilldown, stilldown**

cleartrackingrect

SYNOPSIS *trectnum gstate* **cleartrackingrect** –

Clears the tracking rectangle identified by *trectnum*, as set by **settrackingrect**, in the device referred to by *gstate* (or the current graphics state if *gstate* is **null**). If no such rectangle exists, the **invalidid** error is executed.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **settrackingrect**

composite

SYNOPSIS *srcx srcy width height srcgstate destx desty op* **composite** –

Performs the compositing operation specified by *op* between pairs of pixels in two images, a source and a destination. The source pixels are in the window device referred to by the *srcgstate* graphics state, and the destination pixels are in the current window. If *srcgstate* is **null**, the current graphics state is assumed. If either graphics state doesn't refer to a window device, the **invalidid** error is executed.

The rectangle specified by *srcx*, *srcy*, *width*, and *height* defines the source image. The outline of the rectangle may cross pixel boundaries due to fractional coordinates, scaling, or rotated axes. The pixels included in the source are all those that the outline of the rectangle encloses or enters.

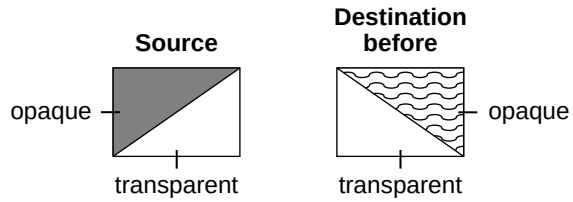
The destination image has the same size, shape, and orientation as the source; *destx* and *desty* give destination's location image compared to the source. (Even if the two graphic states have different orientations, the images will not; **composite** will not rotate images.)

Both images are clipped to the frame rectangles of their respective windows. The destination image is further clipped to the clipping path of the current graphics state. The result of a composite operation replaces the destination image.

op specifies the compositing operation. The choices for *op* and the result of each operation are given in the following illustration.

ERRORS **invalidid**, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **compositerect, setalpha**



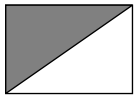
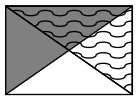
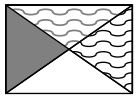
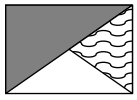
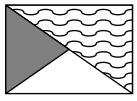
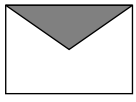
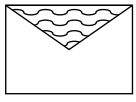
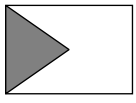
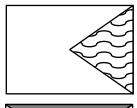
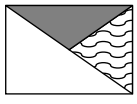
Operation	Destination after	
Copy		Source image.
Clear		Transparent.
PlusD		Sum of source and destination images, with color values approaching 0 as a limit.
PlusL		Sum of source and destination images, with color values approaching 1 as a limit. (PlusL is not implemented for the MegaPixel Display.)
Sover		Source image wherever source image is opaque, and destination image elsewhere.
Dover		Destination image wherever destination image is opaque, and source image elsewhere.
Sin		Source image wherever both images are opaque, and transparent elsewhere.
Din		Destination image wherever both images are opaque, and transparent elsewhere.
Sout		Source image wherever source image is opaque but destination image is transparent, and transparent elsewhere.
Dout		Destination image wherever destination image is opaque but source image is transparent and transparent elsewhere.
Satop		Source image wherever both images are opaque, destination image wherever destination image is opaque but source image is transparent, and transparent elsewhere.

Figure 0-1. Compositing Operations

compositerect

SYNOPSIS *destx desty width height op* **compositerect** –

In general, this operator is the same as the **composite** operator except that there's no real source image. The destination is in the current graphics state; *destx*, *desty*, *width*, and *height* describe the destination image in that graphics state's current coordinate system. The effect on the destination is as if there were a source image filled with the color and coverage specified by the graphics state's current color parameter. *op* has the same meaning as the *op* operand of the **composite** operator; however, one additional operation, Highlight, is allowed.

On the MegaPixel Display, Highlight turns every white pixel in the destination rectangle to light gray and every light gray pixel to white, regardless of the pixel's coverage value. Repeating the same operation reverses the effect. (Highlight may act differently on other devices. For example, on displays that assign just one bit per pixel, it would invert every pixel.)

Note: The Highlight operation doesn't change the value of a pixel's coverage component. To ensure that the pixel's color and coverage combination remains valid, Highlight operations should be temporary and should be reversed before any further compositing.

For **compositerect**, the pixels included in the destination are those that the outline of the specified rectangle encloses or enters. The destination image is clipped to the frame rectangle and clipping path of the window in the current graphics state.

If the current graphics state doesn't refer to a window device, the **invalidid** error is executed.

ERRORS **invalidid**, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **composite, setalpha**

copypage

Warning: This standard PostScript operator has no effect in the OPENSTEP implementation of the Display PostScript system.

countframebuffers

SYNOPSIS – **countframebuffers** *count*

Returns the number of frame buffers that the Window Server is actually using.

ERRORS **stackoverflow**

SEE ALSO **framebuffer**

countscreenlist

SYNOPSIS *context* **countscreenlist** *count*

Returns the number of windows in the screen list that were created by the PostScript context specified by *context*. This is in contrast with **countwindowlist**, which returns the number of windows created by the context without regard to their inclusion in the screen list.

If *context* is 0, all windows in the screen list are counted, without regard to the context that created them.

ERRORS **invalidid**, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **countwindowlist, screenlist, windowlist**

countwindowlist

SYNOPSIS *context* **countwindowlist** *count*

Returns the number of windows that were created by the PostScript context specified by *context*. This is in contrast with **countscreenlist**, which returns the number of windows in the screen list that were created by the PostScript context specified by *context*.

If *context* is 0, all windows are counted, without regard to the context that created them.

ERRORS **stackunderflow**, **typecheck**

SEE ALSO **countscreenlist, screenlist, windowlist**

currentactiveapp

SYNOPSIS – **currentactiveapp** *context*

Warning: Don't use this operator if you're using the Application Kit.

Returns the active application's context. This operator is used by the window packages to assist with wait cursor operation.

ERRORS **stackoverflow**

SEE ALSO **setactiveapp**

currentalpha

SYNOPSIS – **currentalpha** *coverage*

Returns the coverage parameter of the current graphics state.

ERRORS none

SEE ALSO **composite, setalpha**

currentdefaultdepthlimit

SYNOPSIS – **currentdefaultdepthlimit** *depth*

Warning: Don't use this operator if you're using the Application Kit. Use Window's **defaultDepthLimit** class method instead.

Returns the current context's default depth limit. This value determines a new window's depth limit.

ERRORS **stackoverflow**

SEE ALSO **setdefaultdepthlimit**, **setwindowdepthlimit**, **currentwindowdepthlimit**, **currentwindowdepth**

currentdeviceinfo

SYNOPSIS *window* **currentdeviceinfo** *min max iscolor*

Returns device-related information about the current state of *window*. *min* and *max* are the smallest and largest number of bits per sample, respectively, and *iscolor* is a boolean value indicating whether the device is a color device.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

currenteventmask

SYNOPSIS *window* **currenteventmask** *mask*

Warning: Don't use this operator if you're using the Application Kit. Use Window's **eventMask** method instead.

Returns the current Window Server-level event mask for the specified window.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **seteventmask**

currentframebuffertransfer

SYNOPSIS *fbnum* **currentframebuffertransfer** *redproc greenproc blueproc grayproc*

Returns the current transfer functions in effect for the framebuffer indexed by *fbnum*. *fbnum* ranges from 0 to (countframebuffers – 1).

ERRORS **invalidid, stackunderflow, typecheck**

SEE ALSO **setframebuffertransfer, countframebuffers, framebuffer**

currentmouse

SYNOPSIS *window* **currentmouse** *x y*

Warning: Don't use this operator if you're using the Application Kit. Use Window's **getMouseLocation:** instead.

Returns the current x and y coordinates of the mouse location in the base coordinate system of the specified window. If the mouse isn't inside the specified window, these coordinates may be outside the coordinate range defined for the window. If *window* is 0, the current mouse position is returned relative to the screen coordinate system.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **basetocurrent**, **basetoscreen**, **buttondown**, **rightbuttondown**, **rightstilldown**, **setmouse**, **stilldown**

currentowner

SYNOPSIS *window* **currentowner** *context*

Returns a number identifying the PostScript context that currently owns the specified window. By default, this is the PostScript context that created the window.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **setowner**, **termwindow**, **window**

currentshowpageprocedure

SYNOPSIS *window* **currentshowpageprocedure** *proc*

Returns the PostScript procedure that's executed when the **showpage** operator is executed while the specified window is the current device.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **setshowpageprocedure**

currentrusage

SYNOPSIS – **currentrusage** *ctime utime stime msgsend msgrcv nsignals nvcsw nivcsw*

Returns information about the current time of day and about resource usage by the Window Server, as provided by the system call **getrusage()**. The items returned, and their types, are as follows:

Name	Type	Value
ctime	float	Current time in seconds, modulo 10000
utime	float	User time for the Server process in seconds
stime	float	System time for the Server process in seconds
msgsend	int	Messages sent by the Server to clients
msgrcv	int	Message received by the Server from clients
nsignals	int	Number of signals received by the Server process
nvcsw	int	Number of voluntary context switches

Name	Type	Value
nivcsw	int	Number of involuntary context switches

currenttobase

SYNOPSIS *cx cy* **currenttobase** *bx by*

Converts (*cx*,*cy*) from the current coordinate system of the current window to its base coordinate system. If the current device isn't a window, the **invalidid** error is executed.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **basetocurrent**, **basetoscreen**, **currenttoscreen**, **screentobase**, **screentocurrent**

currenttoscreen

SYNOPSIS *cx cy* **currenttoscreen** *sx sy*

Converts (*cx*, *cy*) from the current coordinate system of the current window to the screen coordinate system. If the current device isn't a window, the **invalidid** error is executed.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **basetocurrent**, **basetoscreen**, **currenttobase**, **screentobase**, **screentocurrent**

currentuser

SYNOPSIS – **currentuser** *uid gid*

Returns the user id (*uid*) and the group id (*gid*) of the user currently logged in on the console of the machine that's running the Window Server.

ERRORS **stackoverflow**

currentwaitcursorenabled

SYNOPSIS *context* **currentwaitcursorenabled** *isenabled*

Returns the state of *context*'s wait cursor flag. If *context* is 0, returns the state of the global wait cursor flag.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **setwaitcursorenabled**

currentwindow

SYNOPSIS – **currentwindow** *window*

Returns the window number of the current window. Executes the **invalidid** error if the current device isn't a window.

ERRORS **invalidid**

SEE ALSO **windowdeviceround**

currentwindowalpha

SYNOPSIS *window* **currentwindowalpha** *alpha*

Returns an integer indicating whether the Window Server is currently storing alpha values for the specified window. Possible *alpha* values are:

–2	Window is opaque; alpha values are explicitly allocated.

0	Alpha values are stored explicitly
2	Window is opaque; no explicit alpha

ERRORS **invalidid**, **stackunderflow**, **typecheck**

currentwindowbounds

SYNOPSIS *window* **currentwindowbounds** *x y width height*

Warning: Don't use this operator if you're using the Application Kit. Use Window's **getFrame:** or Application's **getScreenSize:** method instead.

Returns the location and size of the window in screen coordinates. Pass 0 for *window* to get the size of the entire workspace (the smallest rectangle that encloses all active screens).

x and *y* will be in the range [–215, 215 –1]; *width* and *height* will be in the range [0, 10000].

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **movewindow**, **placewindow**

currentwindowdepth

SYNOPSIS *window* **currentwindowdepth** *depth*

Warning: Don't use this operator if you're using the Application Kit.

Returns *window*'s current depth. The **invalidid** error is executed if *window* doesn't exist.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **setwindowdepthlimit**, **currentwindowdepthlimit**, **setdefaultdepthlimit**, **currentdefaultdepthlimit**

currentwindowdepthlimit

SYNOPSIS *window* **currentwindowdepthlimit** *depth*

Warning: Don't use this operator if you're using the Application Kit. Use Window's **depthLimit** method instead.

Returns the window's current depth limit, the maximum depth to which the window can be promoted. Unless altered by the **setwindowdepthlimit** operator, a window's depth limit is equal to its context's default depth limit. The **invalidid** error is executed if *window* doesn't exist.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **setwindowdepthlimit**, **currentwindowdepth**, **setdefaultdepthlimit**, **currentdefaultdepthlimit**

currentwindowdict

SYNOPSIS *window* **currentwindowdict** *dict*

Warning: Don't use this operator if you're using the Application Kit.

Returns the specified window's dictionary.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **setwindowdict**

currentwindowlevel

SYNOPSIS *window* **currentwindowlevel** *level*

Returns *window*'s tier. Executes the **invalidid** error if *window* doesn't exist.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **setwindowlevel**

currentwriteblock

SYNOPSIS – **currentwriteblock** *doesblock*

Returns whether the Window Server delays sending data to a client application whenever the Server's output buffer fills. **currentwriteblock** assumes the current context. If *doesblock* is *true*, the Server waits until the buffer can accept more data. If *doesblock* is *false*, the Server discards data that can't be accepted immediately.

ERRORS **none**

SEE ALSO **setwriteblock**

dissolve

SYNOPSIS *srcx srcy width height srcgstate destx desty delta* **dissolve** –

The effect of this operation is a blending of a source and a destination image. The first seven arguments choose source and destination pixels as they do for **composite**. The exact fraction of the blend is specified by *delta*, which is a floating-point number between 0.0 and 1.0; the resulting image is:

delta *source + (1– *delta*)*destination

If *srcgstate* is null, the current graphics state is assumed. If *srcgstate* or the current graphics state does not refer to a window device, this operator executes the **invalidid** error.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **composite**

dumpwindow

SYNOPSIS *dumplevel window* **dumpwindow** –

Warning: Don't use this operator if you're using the Application Kit.

Prints information about *window* to the standard output file. Only *dumplevel* 0 is implemented. The information printed is the position and number of bytes of backing storage for the window.

ERRORS **invalidid**, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **dumpwindows**

dumpwindows

SYNOPSIS *dumplevel context* **dumpwindows** –

Warning: Don't use this operator if you're using the Application Kit.

Prints information about all windows owned by *context* to the standard output file. If *context* is 0, it prints information about all windows. Only *dumplevel* 0 is implemented.

ERRORS **invalidid**, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **dumpwindow**

erasepage

SYNOPSIS – **erasepage** –

Warning: This standard operator is different in the OPENSTEP implementation.

Erases the entire window to opaque white.

ERRORS **invalidid**

SEE ALSO **copypage**, **showpage**

findwindow

SYNOPSIS *x y place otherwindow* **findwindow** *x' y' window found*

findwindow starts from a given position in the screen list, as explained below, and searches for the first window below that position that contains the point (*x*, *y*). The *x* and *y* values are given in screen coordinates.

The starting position is determined by *place* and *otherwindow*. *place* can be **Above** or **Below**, and *otherwindow* is the window number of a window in the screen list. If you specify **Above 0**, **findwindow** checks all windows in the screen list.

If a window containing the point is found, **findwindow** returns *true*, along with the window number and the corresponding location in the base coordinate system of the window. Otherwise, it returns *false*, and the values of *x'*, *y'*, and *window* are undefined.

ERRORS **rangecheck**, **stackunderflow**, **typecheck**

flushgraphics

SYNOPSIS – **flushgraphics** –

Warning: Don't use this operator if you're using the Application Kit. Use Window's **flushWindow** method instead.

Flushes to the screen all drawing done in the current buffered window. If the current window is retained or nonretained, **flushgraphics** has no effect.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

framebuffer

SYNOPSIS *index string* **framebuffer** *name slot unit romid x y width height maxdepth*

Provides information on the active frame buffer specified by *index*, where *index* ranges from 0 to **countframebuffers**–1. *string* must be large enough to hold the resulting name of the frame buffer. *slot* is the NeXTbus™ slot the frame buffer is physically occupying. If a board supports multiple frame buffers, *unit* uniquely identifies the frame buffer within a slot. The ROM product code is returned in *romid*. The bottom left corner of the frame buffer is returned in *x* and *y* (relative to the screen coordinate system). The size of the frame buffer in pixels is returned in *width* and *height*. *maxdepth* is the maximum depth displayable on this frame buffer (for example, NSTwentyFourBitRGBDepth).

The **limitcheck** error is executed if *string* isn't large enough to hold *name*. The **rangecheck** error is executed if *index* is out of bounds.

ERRORS **limitcheck**, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **countframebuffers**

frontwindow

SYNOPSIS – **frontwindow** *window*

Warning: Don't use this operator if you're using the Application Kit.

Returns the window number of the frontmost window on the screen. If there aren't any windows on the screen, **frontwindow** returns 0.

ERRORS none

SEE ALSO **orderwindow**

hidecursor

SYNOPSIS – **hidecursor** –

Removes the cursor from the screen. It remains in effect until balanced by a call to **showcursor**.

ERRORS none

SEE ALSO **obscurecursor**, **showcursor**

hideinstance

SYNOPSIS *x y width height* **hideinstance** –

In the current window, **hideinstance** removes any instance drawing from the rectangle specified by *x*, *y*, *width*, and *height*. *x*, *y*, *width*, and *height* are given in the window's current coordinate system.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **newinstance**, **setinstance**

image

SYNOPSIS *dict* **image** –

Allows a window's graphics state object to be used as a source of sample data. dict must be an image dictionary in which only those keys listed in the following table are significant:

Key	Type	Value or Meaning
ImageType	integer	(*Required*) Must be 2.

Key	Type	Value or Meaning
XOrigin	real	(*Required*) X origin of the source rectangle in user space coordinates as specified by the transformation in the DataSource entry.
YOrigin	real	(Required) Y origin of the same.
Width	real	(Required) Width of the same.
Height	real	(Required) Height of the same.
ImageMatrix	array	(Required) The transformation matrix.
DataSource	gstate	(Required) A graphics state object that contains the device that will be used as the source of sample data. This device will also be used to determine the pixel representation for the source, and the color space to be used by the image.
Interpolate	boolean	(Optional) Request for image interpolation.
UnpaintedPath	(various)	(Return value) If some of the pixels in the source weren't available (because of clipping), then the UnpaintedPath entry contains a userpath in the current (destination) user space that encloses the area that couldn't be filled.
PixelCopy	boolean	(Optional) If true, indicates that the source pixels should be copied directly, without going through the normal color conversion, transfer function, or halftoning. The bits per pixel of the source must match the bits per pixel of the destination, otherwise a typecheck error will occur. If false or not present, the pixels will be imaged in the usual way.

ERRORS **invalidid, rangecheck, stackunderflow, typecheck**

SEE ALSO **alphaimage**

initgraphics

SYNOPSIS – **initgraphics** –

Warning: This standard operator has additional effects in the OPENSTEP implementation of the Display PostScript system.

In addition to the effects documented by Adobe, this operator sets the coverage parameter in the current window's graphics state to 1 (opaque) and turns off instance drawing

ERRORS none

SEE ALSO **hideinstance**, **newinstance**, **setalpha**, **setinstance**

machportdevice

SYNOPSIS *width height bbox matrix hostname portname pixelencoding* **machportdevice** –

Sets up a PostScript device that can provide a generic rendering service for device-control programs requiring page bitmaps from PostScript documents. For each rendered page, **machportdevice** sends a Mach message containing the page bitmap to a port that has been registered with the name server on the network.

width and *height* are integers that determine the number of device pixels for the page.

bbox is an array of integers that defines the rectangle (by giving its lower left and upper right corners) that encompasses the imageable area. The array takes the form

[*lowerLeftX lowerLeftY upperRightX upperRightY***]**

For the common case where the entire raster is imageable, *bbox* may be expressed as an empty array. **If** *bbox* isn't in the correct form, or if any portion of the rectangle it expresses falls outside [0 0 *width height*], a **rangecheck** results. The bitmap data is stored in x-axis major indexing order. The device coordinate of the lower left corner of the first pixel is (0,0), the coordinate of the next pixel is (1,0) and so on for the entire scanline. Scanlines are long-word aligned.

matrix is the default transformation matrix for the device.

hostname and *portname* are strings that together identify the port that will receive the Mach messages. If *hostname* is empty, the local host is assumed. If it's "*", the port is searched for on all available hosts. If (in any case) the port can't be found, a **rangecheck** results.

pixelencoding is a dictionary describing the format for the image data rendered by the Window Server. It should contain these entries:

Key	Type	Value
samplesPerPixel	integer	Must be 1.
bitsPerSample	integer	Must be 1 or 2.
colorSpace	integer	Color space specification (see below).
isPlanar	boolean	*true* if sample values are stored in separate arrays (currently must be *false*).
defaultHalftone	dictionary	Passed to **sethalftone** during device creation to set up device default halftone.
initialTransfer	procedure	Passed to **settransfer** during device creation to set up the initial transfer function for device.
jobTag	integer	Allows **machportdevice** to tag rendering jobs. This value is included in the **jobTag** field of all printpage messages generated by this device.

The value of **colorSpace** should be one of the following values, predefined in **nextdict**:

Name	Value	Description
NSOneIsBlackColorSpaceNumber	0	Monochromatic, high sample value is black.
NSOneIsWhiteColorSpaceNumber	1	Monochromatic, high sample value is white.
NSRGBColorSpaceNumber	2	RGB, (1,1,1) is white.
NSCMYKColorSpaceNumber	5	CMYK, (0,0,0,0) is white.

Only the following combinations of **colorSpace** and **bitsPerSample** are supported:

colorSpace	bitsPerSample
NSOneIsBlackColorSpaceNumber	1
NSOneIsWhiteColorSpaceNumber	2

These read-only pixel-encoding dictionaries are predefined in **nextdict**:

Name	Description
NeXTLaser-300	NeXT Laser Printer at 300 dpi resolution
NeXTLaser-400	NeXT Laser Printer at 400 dpi resolution
NeXTMegaPixelDisplay	MegaPixel Display's 2 bits-per-pixel gray

The pagebuffer data is passed out-of-line, appearing in the receiving application's address space. (If the receiver is on the same host, the received pagebuffer references the same physical memory as the Window Server's pagebuffer, and is mapped copy-on-write.) The application should use **vm_deallocate()** to release the pagebuffer memory when it's no longer needed. The receiver must acknowledge receipt of the data by sending a simple **msg_header_t** (with **msg_id** == NX_PRINTPAGEMSGID) back to the **remote_port** passed in the print message. The Window Server will not continue executing the page description until acknowledgement is received.

If more than one copy of the page is needed (through either the **copypage** or **#copies** mechanism) each copy is sent as a separate message. In this case the same pagebuffer will be sent in multiple messages. The **letter**, **legal**, and **note** page types are gracefully ignored.

Messaging errors cause the **invalidaccess** error to be executed.

EXAMPLES This example sets up a 400 dpi 8.5 by 11 inch page on a raster with upper left origin (as with the NeXT 400 dpi Laser Printer) and sends its print page messages to the port named "nlp-123" on the local host:

```
/dpi 400 def
/width dpi 8.5 mul cvi def
/height dpi 11 mul cvi def

width height      % page bitmap dimensions in pixels
[]                % use it all
[dpi 72 div 0 0 dpi -72 div 0 height] % device transform
() (nlp-123)      % host (local) & port
NeXTLaser-400     % pixel-encoding description
machportdevice
```

This example sets up an 8 by 10 inch page on the same 8.5 by 11 inch page. It specifies a 400 dpi raster with 1/4 inch horizontal margins and 1/2 inch vertical margins:

```
/dpi 400 def
/width dpi 8.5 mul cvi def
/height dpi 11 mul cvi def
/topdots dpi .5 mul cvi def
/leftdots dpi .25 mul cvi def

width height         % page bitmap dimensions in pixels
[
   leftdots
   topdots
   width leftdots sub
   height topdots sub
]                    % imageable area of bounding box
[
   dpi 72 div
   0
   0
   dpi -72 div
   leftdots
   height topdots sub
]                    % device transform
() (nlp-123)         % host (local) & port
NeXTLaser-400        % pixel-encoding description
machportdevice
```

Note that in this example, the user space origin is at the lower left corner of the imageable area (*leftdots*, *height-topdots*) in the device raster coordinate system. Usually, the imageable area is meant to correspond with the ultimate destination of the bits. For example, a printer may have a constant-sized pagebuffer with a fixed orientation in the paper path, but be able to accept various sizes of paper. In this case, the page bitmap size will always be fixed, but the imageable area and default device transformation can be adjusted to make the user space origin appear at the lower left corner of each printed page.

ERRORS **invalidaccess**, **limitcheck**, **rangecheck**, **stackunderflow**, **typecheck**

movewindow

SYNOPSIS *x y window* **movewindow** –

Warning: Don't use this operator if you're using the Application Kit. Use Window's **moveTo::** method instead.

Moves the lower left corner of the specified window to the screen coordinates (*x*, *y*). No portion of the repositioned window can have an x or y coordinate with an absolute value greater than 16000. The operands can be integer, real, or radix numbers; however, they are converted to integers in the Window Server by rounding toward 0.

The window need not be the frontmost window. This operator doesn't change *window*'s ordering in the screen list.

ERRORS **invalidid**, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **currentwindowbounds**, **placewindow**

newinstance

SYNOPSIS – **newinstance** –

Removes any instance drawing from the current window.

ERRORS **invalidid**

SEE ALSO **hideinstance**, **setinstance**

nextrelease

SYNOPSIS – **nextrelease** *string*

Returns version information about this release.

ERRORS **stackoverflow**

SEE ALSO **osname**, **ostype**

NextStepEncoding

SYNOPSIS – **NextStepEncoding** *array*

Pushes the NextStepEncoding vector on the operand stack. This is a 256-element array, indexed by character codes, whose values are the character names for those codes.

ERRORS **stackoverflow**

obscurecursor

SYNOPSIS – **obscurecursor** –

Removes the cursor image from the screen until the next time the mouse is moved. It's usually called in response to typing by the user, so the cursor won't be in the way. If the cursor has already been removed due to an **obscurecursor** call, **obscurecursor** has no effect.

ERRORS none

SEE ALSO **hidecursor**, **revealcursor**

orderwindow

SYNOPSIS *place otherwindow window* **orderwindow** –

Warning: Don't use this operator if you're using the Application Kit. Use Window's **orderWindow:relativeTo:** instead.

Orders *window* in the screen list as indicated by *place* and *otherwindow*. *place* can be **Above**, **Below**, or **Out**:

- If *place* is **Above** or **Below**, the window is placed in the screen list immediately above or below the window specified by *otherwindow*.
- If *place* is **Above** or **Below** and *otherwindow* is 0, the window is placed above or below all windows in its tier.
- If *place* is **Above** or **Below**, *otherwindow* must be a window in the screen list; otherwise, the **invalidid** error is executed.
- If *place* is **Out**, *otherwindow* is ignored, and the window is removed from the screen list, so it won't appear anywhere on the screen. Windows that aren't in the screen list don't receive user events.

Note: **orderwindow** doesn't change which window is the current window.

ERRORS **invalidid**, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **frontwindow**

osname

SYNOPSIS – **osname** *string*

Returns a string identifying the operating system of the Window Server's current operating environment. **osname** is defined in the **statusdict** dictionary, a dictionary that defines operators specific to a particular implementation of the PostScript language. **osname** can be executed as follows:

```
statusdict /osname get exec
```

ERRORS none

SEE ALSO **nextrelease**, **ostype**

ostype

SYNOPSIS – **ostype** *int*

Returns a number identifying the operating system of the Window Server's current operating environment. **ostype** is defined in the **statusdict** dictionary, a dictionary that defines operators specific to a particular implementation of the PostScript language. **ostype** can be executed as follows:

```
statusdict /ostype get exec
```

ERRORS none

SEE ALSO **nextrelease**, **osname**

placewindow

SYNOPSIS *x y width height window* **placewindow** –

Warning: Don't use this operator if you're using the Application Kit. Use Window's **placeWindow:** method instead.

Repositions and resizes the specified window, effectively allowing it to be resized from any corner or point. *x*, *y*, *width*, and *height* are given in the screen coordinate system. No portion of the repositioned window can have an x or y coordinate with an absolute value greater than 16000; *width* and *height* must be in the range from 0 to 10000. The four operands can be integer or real numbers; however, they are converted to integers in the Window Server by rounding toward 0.

placewindow places the lower left corner of the window at (*x*, *y*) and resizes it to have a width of *width* and a height of *height*. The pixels that are in the intersection of the old and new positions of the window survive unchanged (see Figure 0-2). Any other areas of the newly positioned window are filled with the window's exposure color (see **setexposurecolor**).

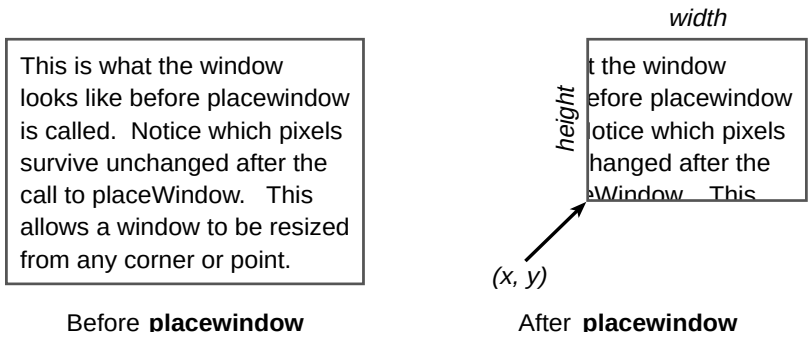


Figure 0-2. **placewindow**

After moving or resizing a window with **placewindow**, you must execute the **initmatrix** and **initclip** operators to reestablish the window's default transformation matrix and default clipping path.

ERRORS **invalidid**, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **currentwindowbounds**, **movewindow**, **setexposurecolor**

playsound

SYNOPSIS *soundname priority* **playsound** –

Plays the sound *soundname*. The Window Server searches for a standard soundfile of the name

*soundname***.snd**

The search progresses through the following directories in the order given, stopping when the sound is located.

~/Library/Sounds
/LocalLibrary/Sounds
/NextLibrary/Sounds

No error occurs if the soundfile isn't found: The operator has no effect.

The soundfile's playback is assigned the priority level *priority*. The playback interrupts any currently playing sound of the same or lower priority level.

ERRORS **stackunderflow**, **typecheck**

posteventbycontext

SYNOPSIS *type x y time flags window subtype misc0 misc1 context* **posteventbycontext** *success*

Posts an event to the specified context. The nine parameters preceding the context parameter coincide with the NXEvent structure members (see "Types and Constants" for the definition of the NXEvent structure). The *x* and *y* coordinate arguments are passed directly to the receiving context without undergoing any transformations. *window* is the Window Server's global window number. Returns *true* if the event was successfully posted to *context*, and *false* otherwise.

You use this operator to post an application-defined event to your own application. Use Mach messaging to communicate between applications.

ERRORS **stackunderflow**, **typecheck**

readimage

SYNOPSIS *x y width height proc0 [... procn-1] string bool* **readimage** –

Reads the pixels that make up the rectangular image described by *x*, *y*, *width*, and *height* in the current window. (Most programmers should use **NXReadBitmap()** instead of this operator.)

Usually the image is the rectangle that has a lower left corner of (*x*, *y*) in the current coordinate system and a width and height of *width* and *height*. If the axes have been rotated so that the sides of the rectangle are no longer aligned with the edges of the screen, the image is the smallest screen-aligned rectangle enclosing the given rectangle.

You typically call **sizeimage** before **readimage** (sending it the same *x*, *y*, *width*, and *height* values you'll use for **readimage**) to find out *ncolors*, the number of color components that **readimage** must read. *bool* is a boolean value that determines whether **readimage** reads the alpha component in addition to the color component(s) for each pixel. The total number of components to be read for each pixel, together with the *multiproc* value returned by **sizeimage**, determine *n*, the number of procedures that **readimage** requires. If *multiproc* is *false*, *n* equals 1. Otherwise, *n* equals the number of color components plus the alpha component, if present.

readimage executes the procedures in order, 0 through *n–1*, as many times as needed. For each execution, it pushes on the operand stack a substring of *string* containing the data from as many scanlines as possible. The length of the substring is a multiple of

width * *bits/sample* * (*samples/proc*) / 8

rounded up to the nearest integer. (The *width* and *bits/sample* values are provided by the **sizeimage** operator. *samples* is the number of color components plus 1 for the alpha component, if present.)

The samples are ordered and packed as they are for the **image**, **colorimage**, or **alphaimage** operator. For example, the alpha component is last and, if necessary, extra bits fill up the last character of every scanline. Note that the contents of *string* are valid only for the duration of one call to one procedure because the same string is reused on each procedure call. The **rangecheck** error is executed if *string* isn't long enough for one scanline.

ERRORS **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **alphaimage**, **sizeimage**

revealcursor

SYNOPSIS – **revealcursor** –

Redisplays the cursor that was hidden by a call to **obscurecursor**, assuming that the cursor hasn't already been revealed by mouse movement. If the cursor hasn't been removed from the screen by a call to **obscurecursor**, **revealcursor** has no effect.

ERRORS none

SEE ALSO **obscurecursor**

rightbuttondown

SYNOPSIS – **rightbuttondown** *isdown*

Returns *true* if the right mouse button is currently down; otherwise it returns *false*.

Note: To test whether the right mouse button is still down from a mouse-down event, use **rightstilldown** instead of **rightbuttondown**; **rightbuttondown** will return *true* even if the mouse button has been released and pressed again since the original mouse-down event.

ERRORS none

SEE ALSO **buttondown**, **currentmouse**, **rightstilldown**, **stilldown**

rightstilldown

SYNOPSIS *eventnum* **rightstilldown** *stilldown*

Returns *true* if the right mouse button is still down from the mouse-down event specified by *eventnum*; otherwise it returns *false*. *eventnum* should be the number stored in the **data** component of the event record for an event of type **Rmousedown**.

ERRORS **stackunderflow**, **typecheck**

SEE ALSO **buttondown**, **currentmouse**, **rightbuttondown**, **stilldown**

screenlist

SYNOPSIS *array context* **screenlist** *subarray*

Fills the array with the window numbers of all windows in the screen list that are owned by the PostScript context specified by *context*. It returns the subarray containing those window numbers, in order from front to back. If *array* isn't large enough to hold them all, this operator will return the frontmost windows that fit in the array.

If *context* is 0, all windows in the screen list are returned.

EXAMPLE `This example yields an array containing the window numbers of all windows in the screen list that are owned by the current PostScript context:`

```
currentcontext
countscreenlist             % find out how many windows
array                       % create array to hold them
currentcontext screenlist   % fill it in
```

ERRORS **invalidaccess**, **invalidid**, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **countscreenlist**, **countwindowlist**, **windowlist**

screentobase

SYNOPSIS *sx sy* **screentobase** *bx by*

Converts (*sx*, *sy*) from the screen coordinate system to the current window's base coordinate system. If the current device isn't a window, the **invalidid** error is executed.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **basetocurrent**, **basetoscreen**, **currenttobase**, **currenttoscreen**, **screentocurrent**

screentocurrent

SYNOPSIS *sx sy* **screentocurrent** *cx* cy

Converts (*sx*,*sy*) from the screen coordinate system to the current coordinate system of the current window. If the current device isn't a window, the **invalidid** error is executed.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **basetocurrent**, **basetoscreen**, **currenttobase**, **currenttoscreen**, **screentobase**

setactiveapp

SYNOPSIS *context* **setactiveapp** –

Warning: Don't use this operator if you're using the Application Kit.

Records the active application's main (usually only) context. **setactiveapp** is used by the window packages to assist with wait cursor operation.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **currentactiveapp**

setalpha

SYNOPSIS *coverage* **setalpha** –

Sets the coverage parameter in the current window's graphics state to *coverage*. *coverage* must be a number between 0 and 1, with 0 corresponding to transparent, 1 corresponding to opaque, and intermediate values corresponding to partial coverage. This establishes how much background shows through for purposes of compositing.

ERRORS **stackunderflow**, **typecheck**, **undefined**

SEE ALSO **composite**, **currentalpha**

setautofill

SYNOPSIS *flag window* **setautofill** –

Applies only to nonretained windows; sets the autofill property of *window* to *the value of flag*. If *true*, newly exposed areas of the window or areas created by **placewindow** will automatically be filled with the window's exposure color. If *false*, these areas will not change (typically they will continue to contain the image of the last window in that area). If the current device is not a window, this operator executes the **invalidid** error.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **placewindow**, **setexposurecolor**, **setsendexposed**

setcursor

SYNOPSIS *x y mx my* **setcursor** –

Sets the cursor image and hot spot. Rather than executing this operator directly, you'd normally use a NXCursor object to define and manage cursors.

A cursor image is derived from a 16-pixel-square image in a window that's generally placed off-screen. The *x* and *y* operands specify the upper left corner of the image in the window's current coordinate system. The *mx* and *my* operands specify the relative offset (in units of the current coordinate system) from (*x*, *y*) to the *hot spot*, the point in the cursor that coincides with the mouse location. Assuming the current coordinate system is the base coordinate system, *mx* must be an integer from 0 to 16, and *my* must be an integer from 0 to –16. After **setcursor** is executed, the image in the window is no longer needed.

The cursor is placed on the screen using Sover compositing. The cursor's opaque areas (alpha = 1) completely cover the background, while its transparent areas (alpha < 1) allow the background to show through to a greater extent depending on the alpha values present in the cursor image.

Note: To make the off-screen window transparent, you can use **compositerect** with **Clear**.

The **rangecheck** error is executed if the image doesn't lie entirely within the specified window or if the point (*mx*, *my*) isn't inside the image. If the current device isn't a window, the **invalidid** error is executed.

ERRORS **invalidid**, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **hidecursor**, **obscurecursor**, **setmouse**

setdefaultdepthlimit

SYNOPSIS *depth* **setdefaultdepthlimit** –

Warning: Don't use this operator if you're using the Application Kit.

Sets the current context's default depth limit to *depth*. The Window Server assigns each new context a default depth limit equal to the maximum depth supported by the system. When a new window is created, its depth limit is set to its context's default depth limit.

These depths are defined in **nextdict**:

Depth	Meaning
NSTwoBitGrayDepth	1 spp, 2bps, 2bpp, planar
NSEightBitGrayDepth	1 spp, 8bps, 8bpp, planar
NSTwelveBitRGBDepth	3 spp, 4bps, 16bpp, interleaved
NSTwentyFourBitRGBDepth	3 spp, 8bps, 32bpp, interleaved

where *spp* is the number of samples per pixel; *bps* is the number of bits per sample; and *bpp* is the number of bits per pixel, also known as the window's depth. (The samples-per-pixel value excludes the alpha sample, if present.) *planar* and *interleaved* refer to how the sample data is configured. If a separate data channel is used for each sample, the configuration is *planar*. If data for all samples is stored in a single data channel, the configuration is *interleaved*.

When an alpha sample is present, the number of bits per pixel doubles for planar configurations (4 for NSTwoBitGrayDepth and 16 for NSEightBitGrayDepth). Interleaved configurations already account for an alpha sample whether or not it's present; thus, the number of bits per pixel for NSTwelveBitRGBDepth and NSTwentyFourBitRGBDepth depths remains unchanged.

The constant NSDefaultDepth is also available. If *depth* is NSDefaultDepth, the context's default depth limit is set to the Window Server's maximum visible depth, which is determined by which screens are active.

The **rangecheck** error is executed if *depth* is invalid.

ERRORS **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **currentdefaultdepthlimit**, **setwindowdepthlimit**, **currentwindowdepthlimit**, **currentwindowdepth**

seteventmask

SYNOPSIS *mask window* **seteventmask** –

Warning: Don't use this operator if you're using the Application Kit. Use Window's **setEventMask:** method instead.

Sets the Server-level event mask for the specified window to *mask*. For windows created by the window packages, this mask may allow additional event types beyond those requested by the application. The following operand names are defined for *mask*:

Mask Operand	Event Type Allowed
Lmousedownmask	Mouse-down, left or only mouse button
Lmouseupmask	Mouse-up, left or only mouse button
Rmousedownmask	Mouse-down, right mouse button
Rmouseupmask	Mouse-up, right mouse button
Mousemovedmask	Mouse-moved
Lmousedraggedmask	Mouse-dragged, left or only mouse button
Rmousedraggedmask	Mouse-dragged, right mouse button
Mouseenteredmask	Mouse-entered
Mouseexitedmask	Mouse-exited
Keydownmask	Key-down
Keyupmask	Key-up
Flagschangedmask	Flags-changed
Kitdefinedmask	Kit-defined
Sysdefinedmask	System-defined
Appdefinedmask	Application-defined
Allevents	All event types

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **currenteventmask**

setexposurecolor

SYNOPSIS – **setexposurecolor** –

Applies to nonretained windows only; sets the exposure color to the color specified by the current color parameter in the current graphics state. The exposure color (white by default) determines the color of newly exposed areas of the window and of new areas created by **placewindow**. The alpha value of these areas is always 1 (opaque). If the current device is not a window, this operator executes the **invalidid** error.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **placewindow**, **setautofill**, **setsendexposed**

setflushexposures

SYNOPSIS *flag* **setflushexposures** –

Warning: Don't use this operator if you're using the Application Kit.

Sets whether window-exposed and screen-changed subevents are flushed to clients. If *flag* is *false*, no window-exposed or screen-changed events are flushed to the client until **setflushexposures** is executed with *flag* equal to *true*. By default, window-exposed and screen-changed events are flushed to clients.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

setframebuffertransfer

SYNOPSIS *redproc greenproc blueproc grayproc fbnum* **setframebuffertransfer** –

Warning: This operator should only be used for the development of screen-calibration products.

Sets the framebuffer transfer functions in effect for the framebuffer indexed by fbnum. fbnum ranges from 0 to countframebuffers–1. The framebuffer transfer describes the relationship between the framebuffer values of the display, and the voltage produced to drive the monitor.

The initial four operands define the transfer procedures: Monochrome devices use grayproc (but see the Note below), color devices use the others. The procedures must be allocated in shared virtual memory. In addition, the Window Server assumes that the framebuffer values are directly proportional to screen brightness. This is important for the accuracy of dithering, compositing, and similar calculations.

The default transfer for NeXT Color Displays is

```
{ 1 2.2 div exp } bind dup dup {}
```

Note: setframebuffertransfer is unsupported on the current generation of NeXT monochrome displays.

It's possible to make framebuffer transfer functions persist beyond the lifetime of the Window Server by storing a property in the NetInfo screens database. In the local NetInfo domain, /localconfig/screens holds the configuration information for the screens known to the Window Server (MegaPixel, NeXTdimension, and so on). These specify the layout and activation state of the screen. The NetInfo defaultTransfer property can contain a string of PostScript code suitable for execution by the setframebuffertransfer operator (without the fbnum parameter). For example, the following represents the NetInfo configuration for a NeXTdimension screen with a default gamma of 2.0:

```
localhost:1#  niutil -read  . /localconfig/screens/NeXTdimension
   name: NeXTdimension
   slot: 2
   unit: 0
   defaultTransfer: {1 2.0 div exp } dup dup dup
   bounds: 0 1120 0 832
   active: 1
   _writers: *
```

The **defaultTransfer** property is used to configure the screen each time the Window Server starts up. This allows monitor calibration products to save their settings so the next time the Window Server starts up, the new values will be used. Note that in some cases, the NetInfo configuration state for a monitor will not have **active** equal to 1, although the monitor is being used by the Window Server. If there are no active screens (screens that are explicitly marked as being active), the Window Server uses a suitable default, however, the other NetInfo properties for that screen are ignored. Thus, you must be sure that the screen for which you are adding a **defaultTransfer** value has **active** set to 1.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **setframebuffertransfer**, **countframebuffers**, **framebuffer**

setinstance

SYNOPSIS *flag* **setinstance** –

Sets the instance-drawing mode in the current graphics state on (if *flag* is *true*) or off (if *flag* is *false*).

ERRORS **stackunderflow**, **typecheck**

SEE ALSO **hideinstance**, **newinstance**

setmouse

SYNOPSIS *x y* **setmouse** –

Moves the mouse location (and, correspondingly, the cursor) to (*x*, *y*), given in the current coordinate system. If the current device isn't a window, the **invalidid** error is executed.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **adjustcursor**, **basetocurrent**, **currentmouse**, **screentocurrent**

setowner

SYNOPSIS *context window* **setowner** –

Sets the owning PostScript context of *window* to *context*. The window is terminated automatically when *context* is terminated.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **currentowner**, **termwindow**, **window**

setsendexposed

SYNOPSIS *flag window* **setsendexposed** –

Warning: Don't use this operator if you're using the Application Kit.

Controls whether the Window Server generates a window-exposed subevent (of the kit-defined event) for *window* under the following circumstances:

- Nonretained window: When an area of the window is exposed, or a new area is created by **placewindow**
- Retained or buffered window: When an area of the window that had instance drawing in it is exposed

By default, window-exposed subevents are generated under these circumstances. In any case, the window-exposed subevent isn't flushed to the application until the Window Server receives another event.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **setflushexposures**, **placewindow**, **setautofill**, **setexposurecolor**

setshowpageprocedure

SYNOPSIS *proc window* **setshowpageprocedure** –

Warning: Don't use this operator if you're using the Application Kit.

Sets the PostScript procedure that's executed, for the specified window, when the **showpage** procedure is executed. *proc* must be allocated in shared virtual memory.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **currentshowpageprocedure**

settrackingrect

SYNOPSIS *x y width height leftbool rightbool insidebool userdata trectnum gstate*
settrackingrect –

SYNOPSIS *or*

SYNOPSIS *x y width height optionarray trectnum gstate* **settrackingrect** –

> **Important:** The settrackingrect operator boasts two form, distinguished by the number and contents of the operands that are passed. The operator itself looks at its operands to determine how to proceed. The common portion of the two forms is described immediately below. Attention is then paid to the features that set the forms apart.

Sets a tracking rectangle in the window referred to by *gstate* to the rectangle specified by *x*, *y*, *width*, and *height* (in the coordinate system of that graphics state). If *gstate* is **null**, the window referred to by the current graphics state is used. *trectnum* is an arbitrary integer that can be any number except 0. It's used to identify tracking rectangles; no two tracking rectangles can share the same *trectnum* value. Any number of tracking rectangles may be set in a single window.

The tracking rectangle will remain in effect until **cleartrackingrect** is called, or until another tracking rectangle with the same *trectnum* is set.

Form 1

SYNOPSIS *x y width height leftbool rightbool insidebool userdata trectnum gstate*
settrackingrect –

In this form, the application receives mouse-exited and mouse-entered events as the cursor leaves and reenters the visible portion of the tracking rectangle. In the event record for a mouse-exited or mouse-entered event, the **data** component will contain *trectnum* along with the event number of the last mouse-down event.

userdata is an arbitrary integer that you assign to the tracking rectangle. Since several tracking rectangles can share the same *userdata* value, you can use *userdata* to identify an object in your application that will be notified when a mouse-entered or mouse-exited event occurs in any of the tracking rectangles.

You can specify that mouse-entered and mouse-exited events be generated only if certain mouse buttons are down. If *leftbool* is *true*, the events will be generated only when the left mouse button is down; likewise for *rightbool* and the right mouse button. If both *leftbool* and *rightbool* are *true*, the events will be generated only if both mouse buttons are down. If both *leftbool* and *rightbool* are *false*, the position of the mouse buttons isn't taken into account in generating mouse-entered and mouse-exited events.

settrackingrect causes the Window Server to repeatedly compare the current cursor position to the previous one to see whether the cursor has moved from inside the tracking rectangle to outside it or vice versa. *insidebool* tells **settrackingrect** whether to consider the initial cursor position to be inside or outside the tracking rectangle:

- If *insidebool* is *true* and the cursor is initially outside the tracking rectangle, a mouse-exited event is generated.
- If *insidebool* is *false* and the cursor is initially inside the tracking rectangle, a mouse-entered event is generated.

Form 2

SYNOPSIS *x y width height optionarray trectnum gstate* **settrackingrect** –

In this form, **settrackingrect** sets special event-gathering attributes of a rectangle (events are *not* generated when the boundary is crossed).

optionarray contains key-value pairs that define the attributes that you're interested in. An empty option array is meaningless and will raise a **rangecheck** error. The following keys are currently defined:

Key	Type	Meaning
Pressure	bool	Treat non-zero pressure values as a mouse-down (*false* by default)
Coalesce	bool	Coalesce mouse motion events (*true* by default)

EXAMPLE This example turns pressure on and coalescing off (thereby switching the default values):

```
0 0 10 10 [/Pressure true /Coalesce false] 1 null settrackingrect
```

ERRORS **invalidid**, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **cleartrackingrect**

setwaitcursorenabled

SYNOPSIS *bool context* **setwaitcursorenabled** –

Allows applications to enable and disable wait cursor operation in the specified context. If *context* is 0, **setwaitcursorenabled** sets the global wait cursor flag, which overrides all per-context settings. If the global flag is set to *false*, the wait cursor is disabled for all contexts.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **currentwaitcursorenabled**

setwindowdepthlimit

SYNOPSIS *depth window* **setwindowdepthlimit** –

Warning: Don't use this operator if you're using the Application Kit. Use Window's **setDepthLimit:** method instead.

Sets the depth limit of *window* to *depth*. These depths are defined in **nextdict**:

Depth	Meaning
NSTwoBitGrayDepth	1 spp, 2bps, 2bpp, planar
NSEightBitGrayDepth	1 spp, 8bps, 8bpp, planar
NSTwelveBitRGBDepth	3 spp, 4bps, 16bpp, interleaved
NSTwentyFourBitRGBDepth	3 spp, 8bps, 32bpp, interleaved

where *spp* is the number of samples per pixel; *bps* is the number of bits per sample; and *bpp* is the number of bits per pixel, also know as the window's depth. (The samples-per-pixel value excludes the alpha sample, if present.) *planar* and *interleaved* refer to how the sample data is configured. If a separate data channel is used for each sample, the configuration is *planar*. If data for all samples is stored in a single data channel, the configuration is *interleaved*.

When an alpha sample is present, the number of bits per pixel doubles for planar configurations (4 for NSTwoBitGrayDepth and 16 for NSEightBitGrayDepth). Interleaved configurations already account for an alpha sample whether or not it's present; thus, the number of bits per pixel for NSTwelveBitRGBDepth and NSTwentyFourBitRGBDepth depths remains unchanged.

Another constant, NSDefaultDepth, is defined as the default depth limit in the Window Server's current context. If *depth* is NSDefaultDepth, then the window's depth limit is set to the context's default depth limit. If the resulting depth is lower than the window's current depth, the window's data is dithered down to this depth, which may result in the loss of graphic information.

The **rangecheck** error is executed if *depth* is invalid. The **invalidid** error is executed if *window* doesn't exist.

ERRORS **invalidid**, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **currentwindowdepthlimit**, **setdefaultdepthlimit**, **currentdefaultdepthlimit**, **currentwindowdepth**

setwindowdict

SYNOPSIS *dict window* **setwindowdict** –

Warning: Don't use this operator if you're using the Application Kit.

Sets the dictionary for *window* to *dict*.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **currentwindowdict**

setwindowlevel

SYNOPSIS *level window* **setwindowlevel** –

Sets the window's tier to that specified by *level*. Window tiers constrain the action of the **orderwindow** operator; see **orderwindow** for more information.

You rarely use this operator. To make a panel float above other windows, use the Panel class's **setFloatingPanel:** method.

Attempting to change the level of **workspaceWindow** executes the **invalidaccess** error. (**workspaceWindow** is a PostScript name whose value is the window number of the workspace window.)

ERRORS **invalidaccess**, **invalidid**, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **currentwindowlevel**, **orderwindow**

setwindowtype

SYNOPSIS *type window* **setwindowtype** –

Warning: Don't use this operator if you're using the Application Kit. Use Window's **setBackingType:** method instead.

Sets the window's buffering type to that specified. Currently, the only allowable type conversions are from Buffered to Retained and from Retained to Buffered. All other possibilities execute the **limitcheck** error.

ERRORS **invalidaccess**, **invalidid**, **limitcheck**, **stackunderflow**, **typecheck**

SEE ALSO **window**

setwriteblock

SYNOPSIS *bool* **setwriteblock** –

Sets how the Window Server responds when its output buffer to a client application fills. If *bool* is *true*, the Server defers sending data (event records, error messages, and so on) to that application until there's once again room in the output buffer. In this way, no output data is lost—this is the Server's default behavior. If *bool* is *false*, the Server ignores the state of the output buffer: If the buffer fills and there's more data to be sent, the new data is lost. **setwriteblock** operates on the current context.

Most programmers won't need to use this operator. If you do use it, make sure that you disable the Window Server's default behavior only during the execution of your own PostScript code. If it's disabled while Application Kit code is being executed, errors will result.

ERRORS **stackoverflow**, **typecheck**

SEE ALSO **currentwriteblock**

showcursor

SYNOPSIS – **showcursor** –

Restores the cursor to the screen if it's been hidden with **hidecursor**, unless an outer nested **hidecursor** is still in effect (because it hasn't yet been balanced by a **showcursor**). For example:

```
% cursor is showing initially
. . .
hidecursor       % hides the cursor
. . .
   hidecursor    % cursor stays hidden
   . . .
   showcursor    % cursor still hidden due to first hidecursor
. . .
showcursor       % displays the cursor
```

ERRORS none

SEE ALSO **hidecursor**

showpage

SYNOPSIS – **showpage** –

Warning: This standard operator is different in the OPENSTEP implementation of the Display PostScript system.

This has no effect if the current device is a window; otherwise, it functions as documented by Adobe.

ERRORS none

SEE ALSO **copypage**, **erasepage**

sizeimage

SYNOPSIS *x y width height matrix* **sizeimage** *pixelswide pixelshigh bits/sample matrix multiproc ncolors*

Returns various parameters required by the **readimage** operator when reading the image contained in the rectangle given by *x*, *y*, *width*, and *height* in the current window. (See **readimage** for more information.)

pixelswide and *pixelshigh* are the width and height of the image in pixels. The operand *matrix* is filled with the transformation matrix from user space to the image coordinate system and pushed back on the operand stack.

The other results of this operator describe the window device and are dependent on the window's depth. Each pixel has *ncolors* color components plus one alpha component; the value of each component is described by *bits/sample* bits. If *multiproc* is *true*, **readimage** will need multiple procedures to read the values of the image's pixels. Here are the values that **sizeimage** returns for windows of various depths:

Window Depth	ncolors	bits/sample	multiproc
NSTwoBitGrayDepth	1	2	*true*
NSEightBitGrayDepth	1	8	*true*
NSTwelveBitRGBDepth	3	4	*false*
NSTwentyFourBitRGBDepth	3	8	*false*

ERRORS **stackunderflow**, **typecheck**

SEE ALSO **alphaimage**, **readimage**

stilldown

SYNOPSIS *eventnum* **stilldown** *stilldown*

Returns *true* if the left or only mouse button is still down from the mouse-down event specified by *eventnum*; otherwise it returns *false*. *eventnum* should be the number stored in the **data** component of the event record for an event of type **Lmousedown**.

ERRORS **stackunderflow**, **typecheck**

SEE ALSO **buttondown**, **currentmouse**, **rightbuttondown**, **rightstilldown**

termwindow

SYNOPSIS *window* **termwindow** –

Warning: Don't use this operator if you're using the Application Kit. Use Window's **close** method instead.

Marks *window* for destruction. If the window is in the screen list, it's removed from the screen list and the screen. The given window number will no longer be valid; any attempt to use it will execute the **invalidid** error. The window will actually be destroyed and its storage reclaimed only after the last reference to it from a graphics state is removed. This can be done by resetting the device in the graphics state to another window or to the null device.

Note: After you use the **termwindow** operator, if the terminated window had been the current window, you should use the **nulldevice** operator to remove references to it.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **window**, **windowdevice**, **windowdeviceround**

window

SYNOPSIS *x y width height type* **window** *window*

Warning: Don't use this operator if you're using the Application Kit. Create a Window object instead.

Creates a window that has a lower left corner of (*x*, *y*) and the indicated width and height. *x*, *y*, *width*, and *height* are given in the screen coordinate system. No portion of a window can have an x or y coordinate with an absolute value greater than 16000; *width* and *height* must be in the range from 0 to 10000. Exceeding these limits executes the **rangecheck** error. The four operands can be integer or real numbers; however, they are converted to integers in the Window Server by rounding toward 0. This operator returns the new window's window number, a nonzero integer that's used to refer to the window.

type specifies the window's buffering type as **Buffered**, **Retained**, or **Nonretained**.

The new window won't be in the screen list; you can put it there with the **orderwindow** operator. Windows that aren't in the screen list don't appear on the screen and don't receive user events.

The **window** operator also does the following:

- Sets the origin of the window's base coordinate system to the lower left corner of the window
- Sets the window's clipping path to the outer edge of the window
- Fills the window with opaque white and sets the window's exposure color to white

Note: This operator does not make the new window the current window; to do that, use **windowdeviceround** or **windowdevice**.

ERRORS **invalidid**, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO **setexposurecolor**, **termwindow**, **windowdeviceround**

windowdevice

SYNOPSIS *window* **windowdevice** –

Sets the current device of the current graphics state to the given window device. It also sets the origin of the window's default matrix to the lower left corner of the window. One unit in the user coordinate system is made equal to 1/72 of an inch. The clipping path is reset to a rectangle surrounding the window. Other elements of the graphics state remain unchanged. This matrix becomes the default matrix for the window: **initmatrix** will reestablish this matrix.

windowdevice is rarely used in OPENSTEP since the coordinate system it establishes isn't aligned with the pixels on the screen. Use the related operator **windowdeviceround** to create a coordinate system that is aligned.

Don't use this operator lightly, as it creates a new matrix and clipping path. It's significantly more expensive than a **setgstate** operator.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **windowdeviceround**

windowdeviceround

SYNOPSIS *window* **windowdeviceround** –

Sets the current device of the current graphics state to the given window device. It also sets the origin of the window's default matrix to the lower left corner of the window. One unit in the user coordinate system is made equal to the width of one pixel. The clipping path is reset to a rectangle surrounding the window. Other elements of the graphics state remain unchanged. This matrix becomes the default matrix for the window: **initmatrix** will reestablish this matrix.

Don't use this operator blithely, as it creates a new matrix and clipping path. It's significantly more expensive than a **setgstate** operator.

ERRORS **invalidid**, **stackunderflow**, **typecheck**

SEE ALSO **windowdevice**

windowlist

SYNOPSIS *array context* **windowlist** *subarray*

Fills the array with the window numbers of all windows that are owned by the PostScript context specified by *context*. It returns the subarray containing those window numbers, in order from front to back. If *array* isn't large enough to hold them all, this operator returns the frontmost windows that fit in the array.

EXAMPLE This example yields an array containing the window numbers of all windows that are owned by the current PostScript context:

```
currentcontext
countwindowlist             % find out how many windows
array                       % create array to hold them
currentcontext windowlist   % fill it in
```

ERRORS **stackunderflow**, **typecheck**

SEE ALSO **countscreenlist**, **countwindowlist**, **screenlist**

NSChangeSpelling

Adopted By: NSText

Declared In: AppKit/NSSpellProtocol.h

Protocol Description

This protocol is implemented by objects in the responder chain that can correct a misspelled word. See the description of the NSSpellChecker class for more information.

Instance Methods

changeSpelling:

– (void)**changeSpelling:**(id)*sender*

Implement this method to replace the selected word in the receiver with a corrected version from the Spelling panel. This message is sent by the NSSpellChecker to the object whose text is being checked. To get the corrected spelling, ask the sender for the string value of its selected cell (visible to the user as the text field in the Spelling panel). This method should replace the selected portion of the text with the string that it gets from the NSSpellChecker.

NSColorPickingCustom

Adopted By: NSColorPicker

Declared In: AppKit/NSColorPicking.h

Protocol Description

Together with the NSColorPickingDefault protocol, NSColorPickingCustom provides a way to add color pickers—custom user interfaces for color selection—to an application's NSColorPanel. The NSColorPickingDefault protocol provides basic behavior for a color picker. The NSColorPicker class adopts the NSColorPickingDefault protocol. The easiest way to implement a color picker is to create a subclass of NSColorPicker and implement the NSColorPickingCustom protocol for this new class.

Note: All of NSColorPickingCustom's methods must be implemented by the custom color picker.

Here are the standard color picking modes and mode constants (defined in **AppKit/NSColorPanel.h**):

Mode	Color Mode Constant
Grayscale-Alpha	NSGrayModeColorPanel
Red-Green-Blue	NSRGBModeColorPanel
Cyan-Yellow-Magenta-Black	NSCMYKModeColorPanel
Hue-Saturation-Brightness	NSHSBModeColorPanel
Custom palette	NSCustomPaletteModeColorPanel
Custom color list	NSColorListModeColorPanel
Color wheel	NSWheelModeColorPanel

In grayscale-alpha, red-green-blue, cyan-magenta-yellow-black, and hue-saturation-brightness modes, the user adjusts colors by manipulating sliders. In the custom palette mode, the user can load an NSImage file (TIFF or EPS) into the NSColorPanel, then select colors from the image. In custom color list mode, the user can create and load lists of named colors. The two custom modes provide NSPopUpLists for loading and saving files. Finally, color wheel mode provides a simplified control for selecting colors.

If your color picker includes submodes, you should define a unique value for each submode. As an example, the slider picker has four values defined in the above list (NSGrayModeColorPanel,

NSRGBModeColorPanel, NSCMYKModeColorPanel, and NSHSBModeColorPanel)—one for each of its submodes.

Method Types

Setting the Current Color

– setColor:

Getting the Mode

– currentMode
– supportsMode:

Getting the View

– provideNewView:

Instance Methods

currentMode

– (int)**currentMode**

Returns the color picker's current mode (or submode, if applicable). The returned value should be unique to your color picker. See this protocol description's list of the unique values for the standard color pickers used by the Application Kit.

See also: **– supportsMode:**

provideNewView:

– (NSView *)**provideNewView:**(BOOL)*initialRequest*

Returns the view containing the color picker's user interface. This message is sent to the color picker whenever the color panel attempts to display it. This may be when the panel is first presented, when the user switches pickers, or when the picker is switched through API. The argument *initialRequest* is YES only when this method is first invoked for your color picker. If *initialRequest* is YES, the method should perform any initialization required (such as lazily loading a nib file, initializing the view, or performing any other custom initialization required for your picker). The NSView returned by this method should be set to automatically resize both its width and height.

setColor:

– (void)**setColor:**(NSColor *)*color*

Adjusts the color picker to make *color* the currently selected color. This method is invoked on the current color picker each time NSColorPanel's **setColor:** method is invoked. If *color* is actually different from the color picker's color (as it would be if, for example, the user dragged a color into NSColorPanel's color well), this method could be used to update the color picker's color to reflect the change.

supportsMode:

– (BOOL)**supportsMode:**(int)*mode*

Returns whether or not the receiver supports the specified picking mode. This method is invoked when the NSColorPanel's is first initialized: It is used to attempt to restore the user's previously selected mode. It is also invoked by NSColorPanel's **setMode:** method to find the color picker that supports a particular mode. See this protocol description's list of the unique mode values for the standard color pickers used by the Application Kit.

See also: **– currentMode**

NSColorPickingDefault

Adopted By: NSColorPicker

Declared In: AppKit/NSColorPicking.h

Protocol Description

The NSColorPickingDefault protocol, together with the NSColorPickingCustom protocol, provides an interface for adding color pickers—custom user interfaces for color selection—to an application's NSColorPanel. The NSColorPickingDefault protocol provides basic behavior for a color picker. The NSColorPickingCustom protocol provides implementation-specific behavior.

The NSColorPicker class implements the NSColorPickingDefault protocol. The simplest way to implement your own color picker is to create a subclass of NSColorPicker, implementing the NSColorPickingCustom protocol for that subclass. However, it's possible to create a subclass of another class, such as NSView, and use it as a base upon which to add the methods of both NSColorPickingDefault and NSColorPickingCustom.

Color Picker Bundles

A class that implements the NSColorPickingDefault and NSColorPickingCustom protocols needs to be compiled and linked in an application's object file. However, your application need not explicitly create an instance of this class. Instead, your application's file package should include a directory named **ColorPickers**; within this directory you should place a directory *MyPickerClass***.bundle** for each custom color picker your application implements. This bundle should contain all resources required for your color picker: nib files, TIFF files, and so on.

NSColorPanel will allocate and initialize an instance of each class for which a bundle is found in the **ColorPickers** directory. The class name is assumed to be the bundle directory name minus the **.bundle** extension.

Color Picker Buttons

NSColorPanel lets the user select a color picker from an NSMatrix of NSButtonCells. This protocol includes methods for providing and manipulating the image that gets displayed on the button.

Color Mask and Color Modes

The color mask determines which color mode is enabled for NSColorPanel. This mask is set before you initialize a new instance of NSColorPanel. NSColorPanelAllModesMask represents the logical OR of the

other color mask constants: It causes the NSColorPanel to display all standard color pickers. When initializing a new instance of NSColorPanel, you can logically OR any combination of color mask constants to restrict the available color modes. The predefined color mask constants are:

Mode	Color Mask Constant
Grayscale-Alpha	NSColorPanelGrayModeMask
Red-Green-Blue	NSColorPanelRGBModeMask
Cyan-Yellow-Magenta-Black	NSColorPanelCMYKModeMask
Hue-Saturation-Brightness	NSColorPanelHSBModeMask
Custom palette	NSColorPanelCustomPaletteModeMask
Custom color list	NSColorPanelColorListModeMask
Color wheel	NSColorPanelWheelModeMask
All of the above	NSColorPanelAllModesMask

When an application's instance of NSColorPanel is masked for more than one color mode, your program can set its active mode by invoking the **setMode:** method with a color mode constant as its argument; the user can set the mode by clicking buttons on the panel. Here are the standard color modes and mode constants:

Mode	Color Mode Constant
Grayscale-Alpha	NSGrayModeColorPanel
Red-Green-Blue	NSRGBModeColorPanel
Cyan-Yellow-Magenta-Black	NSCMYKModeColorPanel
Hue-Saturation-Brightness	NSHSBModeColorPanel
Custom palette	NSCustomPaletteModeColorPanel
Custom color list	NSColorListModeColorPanel
Color wheel	NSWheelModeColorPanel

In grayscale-alpha, red-green-blue, cyan-magenta-yellow-black, and hue-saturation-brightness modes, the user adjusts colors by manipulating sliders. In the custom palette mode, the user can load an NSImage file (TIFF or EPS) into the NSColorPanel, then select colors from the image. In custom color list mode, the user can create and load lists of named colors. The two custom modes provide NSPopUpLists for loading and saving files. Finally, color wheel mode provides a simplified control for selecting colors.

These constants are defined in **AppKit/NSColorPanel.h**.

Method Types

Initializing a Color Picker
– initWithPickerMask:colorPanel:

Setting the Mode
– setMode:

Using Color Lists
– attachColorList:
– detachColorList:

Adding Button Images
– insertNewButtonImage:in:
– provideNewButtonImage

Showing Opacity Controls
– alphaControlAddedOrRemoved:

Responding to a Resized View
– viewSizeChanged:

Instance Methods

alphaControlAddedOrRemoved:

– (void)**alphaControlAddedOrRemoved:**(id)*sender*

Sent by the color panel when the opacity controls have been hidden or displayed. Invoked automatically when the NSColorPanel's opacity slider is added or removed; you never invoke this method directly.

If the color picker has its own opacity controls, it should hide or display them, depending on whether the sender's **showsAlpha** method returns NO or YES.

attachColorList:

– (void)**attachColorList:**(NSColorList *)*colorList*

Tells the color picker to attach the given *colorList*, if it isn't already displaying the list. You never invoke this method; it's invoked automatically by the NSColorPanel when its **attachColorList:** method is invoked. Since NSColorPanel's list mode manages NSColorLists, this method need only be implemented by a custom color picker that manages NSColorLists itself. This method ordinarily doesn't do anything, since NSColorPanel's list mode manages NSColorLists.

See also: **– detachColorList:**

detachColorList:

– (void)**detachColorList:**(NSColorList *)*colorList*

Tells the color picker to detach the given *colorList*, unless the receiver isn't displaying the list. You never invoke this method; it's invoked automatically by the NSColorPanel when its **detachColorList:** method is invoked. Since NSColorPanel's list mode manages NSColorLists, this method need only be implemented by a custom color picker that manages NSColorLists itself. This method ordinarily doesn't do anything, since NSColorPanel's list mode manages NSColorLists.

See also: **– attachColorList:**

initWithPickerMask:colorPanel:

– (id)**initWithPickerMask:**(int)*mask*
colorPanel:(NSColorPanel *)*owningColorPanel*

Notifies the color picker of the color panel's mask and initializes the color picker. This method is sent by the NSColorPanel to all implementors of the color picking protocols when the application's color panel is first initialized. In order for your color picker to receive this message, it must have a bundle in your application's "ColorPickers" directory (described in "Color Picker Bundles" in the Protocol Description).

mask is determined by the argument to the NSColorPanel method **setPickerMask:**. If no mask has been set, *mask* is NSColorPanelAllModesMask. If your color picker supports any additional modes, you should invoke the **setPickerMask:** method when your application initializes to notify the NSColorPanel class. The standard mask constants are:

Mode	Color Mask Constant
Grayscale-Alpha	NSColorPanelGrayModeMask
Red-Green-Blue	NSColorPanelRGBModeMask

Mode	Color Mask Constant
Cyan-Yellow-Magenta-Black	NSColorPanelCMYKModeMask
Hue-Saturation-Brightness	NSColorPanelHSBModeMask
Custom palette	NSColorPanelCustomPaletteModeMask
Custom color list	NSColorPanelColorListModeMask
Color wheel	NSColorPanelWheelModeMask
All of the above	NSColorPanelAllModesMask

This method should examine the mask and determine whether it supports any of the modes included there. You may also check the value in *mask* to enable or disable any subpickers or optional controls implemented by your color picker. Your color picker may also retain *owningColorPanel* in an instance variable for future communication with the color panel.

This method is provided to initialize your color picker; however, much of a color picker's initialization may be done lazily through the NSColorPickingCustom protocol's **provideNewView:** method. If your color picker responds to any of the modes represented in *mask*, it should perform its initialization and return **self**. Color pickers that do so have their buttons inserted in the color panel and continue to receive messages from the panel as the user manipulates it. If the color picker doesn't respond to any of the modes represented in mask, it should do nothing and return **nil**.

See also: **+ setPickerMask:** (NSColorPanel class)

insertNewButtonImage:in:

– (void)**insertNewButtonImage:**(NSImage *)*newButtonImage*
in:(NSButtonCell *)*buttonCell*

Sets *newButtonImage* as *buttonCell*'s image. *buttonCell* is the NSButtonCell object that lets the user choose the picker from the color panel—the color picker's representation in the NSColorPanel's picker NSMatrix. This method should perform application-specific manipulation of the image before it's inserted and displayed by the button cell.

See also: **– provideNewButtonImage**

provideNewButtonImage

– (NSImage *)**provideNewButtonImage**

Returns the image for the mode button that the user uses to select this picker in the color panel, that is, the color picker's representation in the NSColorPanel's picker NSMatrix. (This is the same image that the color panel uses as an argument when sending the **insertNewButtonImage:in:** message.)

setMode:

– (void)**setMode:**(int)*mode*

Sets the color picker's mode. This method is invoked by NSColorPanel's **setMode:** method to ensure that the color picker reflects the current mode. For example, invoke this method during color picker initialization to ensure that all color pickers are restored to the mode the user left them in the last time an NSColorPanel was used.

Most color pickers have only one mode, and thus don't need to do any work in this method. An example of a color picker that uses this method is the slider picker, which can choose from one of several submodes depending on the value of *mode*. The available modes are:

Mode	Color Mode Constant
Grayscale-Alpha	NSGrayModeColorPanel
Red-Green-Blue	NSRGBModeColorPanel
Cyan-Yellow-Magenta-Black	NSCMYKModeColorPanel
Hue-Saturation-Brightness	NSHSBModeColorPanel
Custom palette	NSCustomPaletteModeColorPanel
Custom color list	NSColorListModeColorPanel
Color wheel	NSWheelModeColorPanel

viewSizeChanged:

– (void)**viewSizeChanged:**(id)*sender*

Tells the color picker when the NSColorPanel's view size changes in a way that might affect the color picker. *sender* is the NSColorPanel that contains the color picker. Use this method to perform special preparation when resizing the color picker's view. Since this method is invoked only as appropriate, it's

better to implement this method than to override the method **superviewSizeChanged:** for the NSView in which the color picker's user interface is contained.

See also: **– provideNewView:** (NSColorPickingCustom protocol)

NSComboBoxCellDataSource

(Informal Protocol)

Category Of: NSObject

Declared In: AppKit/NSComboBoxCell.h

Category Description

The NSComboBoxCellDataSource category declares the methods that an NSComboBoxCell uses to access the contents of its data source object. The combo box cell determines how many items to display by sending a **numberOfItemsInComboBoxCell:** message, and accesses individual values with the **comboBoxCell: objectValueForItemAtIndex:** method. Incremental searches—performed when a user types into the combo box's text field while the pop-up list is displayed—are performed by sending **comboBoxCell: indexOfItemWithStringValue:** messages to the combo box cell's data source.

The NSComboBoxCell treats objects provided by its data source as values to be displayed in the combo box's pop-up list. If these objects aren't of common value classes—such as NSString, NSNumber, and so on—you'll need to create a custom NSFormatter to display them. See the NSFormatter class specification for more information.

When an NSComboBoxCellDataSource is asked to supply a data item, the NSComboBoxCell that sends the request is provided as a parameter. This allows a single data source object to manage several sets of data, choosing the appropriate set based on the identify of the NSComboBoxCell that sends the message.

Instance Methods

comboBoxCell:indexOfItemWithStringValue:

– (unsigned int)**comboBoxCell:**(NSComboBoxCell *)*aComboBoxCell*
indexOfItemWithStringValue:(NSString *)*aString*

An NSComboBoxCell uses this method to perform incremental—or "smart"—searched when the user types into the text field with the pop-up list displayed. Your implementation of this method should return the index for the item which matches *aString*, or NSNotFound if no item matches. This method is optional; if you don't provide an implementation for this method, no searches occur.

comboBoxCell:objectValueForItemAtIndex:

– (id)**comboBoxCell:**(NSComboBoxCell *)*aComboBoxCell* **objectValueForItemAtIndex:**(int)*index*

Implement this method to return the object that corresponds to the item at *index* in *aComboBoxCell*. Your data source must implement this method.

numberOfItemsInComboBoxCell:

– (int)**numberOfItemsInComboBoxCell:**(NSComboBoxCell *)*aComboBoxCell*

Implement this method to return the number of items managed for *aComboBoxCell* by your data source object. An NSComboBoxCell uses this method to determine how many items it should display in its pop-up list. Your data source must implement this method.

NSComboBoxDataSource

(Informal Protocol)

Category Of: NSObject

Declared In: AppKit/NSComboBox.h

Category Description

The NSComboBoxDataSource category declares the methods that an NSComboBox uses to access the contents of its data source object. The combo box determines how many items to display by sending a **numberOfItemsInComboBox:** message, and accesses individual values with the **comboBox: objectValueForItemAtIndex:** method. Incremental searches—performed when a user types into the combo box's text field while the pop-up list is displayed—are performed by sending **comboBox: indexOfItemWithStringValue:** messages to the combo box's data source.

The NSComboBox treats objects provided by its data source as values to be displayed in the combo box's pop-up list. If these objects aren't of common value classes—such as NSString, NSNumber, and so on—you'll need to create a custom NSFormatter to display them. See the NSFormatter class specification for more information.

When an NSComboBoxDataSource is asked to supply a data item, the NSComboBox that sends the request is provided as a parameter. This allows a single data source object to manage several sets of data, choosing the appropriate set based on the identify of the NSComboBox that sends the message.

Instance Methods

comboBox:indexOfItemWithStringValue:

– (unsigned int)**comboBox:**(NSComboBox *)*aComboBox* **indexOfItemWithStringValue:** (NSString *)*aString*

An NSComboBox uses this method to perform incremental—or "smart"—searched when the user types into the text field with the pop-up list displayed. Your implementation of this method should return the index for the item which matches *aString*, or NSNotFound if no item matches. This method is optional; if you don't provide an implementation for this method, no searches occur.

comboBox:objectValueForItemAtIndex:

– (id)**comboBox:**(NSComboBox *)*aComboBox* **objectValueForItemAtIndex:**(int)*index*

Implement this method to return the object that corresponds to the item at *index* in *aComboBox*. Your data source must implement this method.

numberOfItemsInComboBox:

– (int)**numberOfItemsInComboBox:**(NSComboBox *)*aComboBox*

Implement this method to return the number of items managed for *aComboBox* by your data source object. An NSComboBox uses this method to determine how many items it should display in its pop-up list. Your data source must implement this method.

NSDPSContextNotification

Adopted By: no OpenStep classes

Declared In: AppKit/NSDPSContext.h

Protocol Description

The NSDPSContextNotification protocol supplies information about the execution status of a sequence of PostScript commands previously sent to the Display PostScript server.

Instance Methods

contextFinishedExecuting:

– (void)**contextFinishedExecuting:**(NSDPSContext *)*context*

Notifies the receiver that the context has finished executing a batch of PostScript commands. See **notifyObjectWhenFinishedExecuting:** (NSDPSContext).

NSDraggingDestination

(informal protocol)

Category Of: NSObject

Declared In: AppKit/NSDragging.h

Description

The NSDraggingDestination informal protocol declares methods that the destination (or recipient) of a dragged image must implement. The destination automatically receives NSDraggingDestination messages as an image enters, moves around inside, and then exits or is released within the destination's boundaries.

In the text here and in the other dragging protocol descriptions, the term *dragging session* is the entire process during which an image is selected, dragged, released, and absorbed or rejected by the destination. A *dragging operation* is the action that the destination takes in absorbing the image when it's released. The *dragging source* is the object that "owns" the image that's being dragged. It's specified as an argument to the **dragImage:at:offset:event:pasteboard:source:slideBack:** message, sent to a window or view object, that instigated the dragging session.

The Dragged Image

The image that's dragged in an image-dragging session is simply an image that represents data that resides on the pasteboard. Although a dragging destination can access the image (through the **draggedImage** method described in the NSDraggingInfo protocol), its primary concern is with the pasteboard data that the image represents—the dragging operation that a destination ultimately performs is on the pasteboard data, not on the image itself.

Valid Destinations

Dragging is a visual phenomenon. To be an image-dragging destination, an object must represent a portion of screen real estate; thus, only window and view objects can be destinations. Furthermore, you must register the pasteboard types that the object will accept by sending the object a **registerForDraggedTypes:** message, defined in both NSWindow and NSView. During a dragging session, a candidate destination only receives NSDraggingDestination messages if the destination is registered for a pasteboard type that matches the type of the pasteboard data being dragged. See the NSPasteboard class specification for more information about pasteboard types.

Although NSDraggingDestination is declared as an informal protocol, the NSWindow and NSView subclasses that you create to adopt the protocol need only implement those methods that are pertinent. (The NSWindow and NSView classes provide private implementations for all of the methods.) Either a window

object or its delegate may implement these methods; however, the delegate's implementation takes precedence if there are implementations in both places.

The Sender of Destination Messages

Each of the NSDraggingDestination methods sports a single argument: *sender*, the object that invoked the method. Within its implementations of the NSDraggingDestination methods, the destination can send NSDraggingInfo protocol messages to *sender* to get more information on the current dragging session.

The Order of Destination Messages

The six NSDraggingDestination methods are invoked in a distinct order:

- As the image is dragged into the destination's boundaries, the destination is sent a **draggingEntered:** message
- While the image remains within the destination, a series of **draggingUpdated:** messages are sent.
- If the image is dragged out of the destination, **draggingExited:** is sent and the sequence of NSDraggingDestination messages stops. If it re-enters, the sequence begins again (with a new **draggingEntered:** message).
- When the image is released, it either slides back to its source (and breaks the sequence) or a **prepareForDragOperation:** message is sent to the destination, depending on the value returned by the most recent invocation of **draggingEntered:** or **draggingUpdated:**.
- If the **prepareForDragOperation:** message returned YES, a **performDragOperation:** message is sent.
- Finally, if **performDragOperation:** returned YES, **concludeDragOperation:** is sent.

Method Types

Before the image is released

– draggingEntered:
– draggingUpdated:
– draggingExited:

After the image is released

– prepareForDragOperation:
– performDragOperation:
– concludeDragOperation:

Instance Methods

concludeDragOperation:

– (void)**concludeDragOperation:**(id <NSDraggingInfo>)*sender*

Invoked when the dragging operation is complete and the previous **performDragOperation:** returned YES. The destination implements this method to perform any tidying up that it needs to do, such as updating its visual representation now that it has incorporated the dragged data. This is the last message that's sent from *sender* to the destination during a dragging session.

draggingEntered:

– (unsigned int)**draggingEntered:**(id <NSDraggingInfo>)*sender*

Invoked when a dragged image enters the destination. Specifically, this method is invoked when the mouse pointer enters the destination's bounds rectangle (if it's a view object) or its frame rectangle (if it's a window object).

This method must return a value that indicates which dragging operation the destination will perform when the image is released. In deciding which dragging operation to return, the method should evaluate the overlap between both the dragging operations allowed by the source (accessible through the **draggingSourceOperationMask** method) and the dragging operations and pasteboard data types that the destination itself supports. The returned value should be exactly one of the following:

Option	Meaning
NSDragOperationCopy	The data represented by the image will be copied.
NSDragOperationLink	The data will be shared.
NSDragOperationGeneric	The operation will be defined by the destination.
NSDragOperationPrivate	The operation is negotiated privately between the source and the destination.
NSDragOperationAll	Combines all the above.

If none of the operations is appropriate, this method should return NSDragOperationNone (this is the default response if the method isn't implemented by the destination).

The code below is a simple example of a method that responds distinctly when one of two different types of data is dragged into the destination view or window. If the dragged data is a color and the source object permits copying, the return value indicates that the destination will permit copying of the color data on the pasteboard. If the dragged data is an RTF file and the source object permits linking, the return value

indicates that the destination will permit linking of the RTF file on the pasteboard. Otherwise the code returns NSDragOperationNone, indicating that the destination will not permit any dragging operations with the data on pasteboard.

```
- (unsigned int)draggingEntered:(id <NSDraggingInfo>)sender
{
    NSPasteboard *pboard;
    NSDragOperation sourceDragMask;

    sourceDragMask = [sender draggingSourceOperationMask];
    pboard = [sender draggingPasteboard];

    if ([[pboard types] indexOfObject:NSColorPboardType] != NSNotFound) {
        if (sourceDragMask & NSDragOperationCopy) {
            return NSDragOperationCopy;
        }
    }
    if ([[pboard types] indexOfObject:NSRTFPboardType] != NSNotFound) {
        if (sourceDragMask & NSDragOperationLink) {
            return NSDragOperationLink;
        }
    }
    return NSDragOperationNone;
}
```

See also: – **draggingUpdated:**, – **draggingExited:**, – **prepareForDragOperation:**

draggingExited:

– (void)**draggingExited:**(id <NSDraggingInfo>)*sender*

Invoked when the dragged image exits the destination's bounds rectangle (in the case of a view object) or its frame rectangle (in the case of a window object).

draggingUpdated:

– (unsigned int)**draggingUpdated:**(id <NSDraggingInfo>)*sender*

Invoked periodically as the image is held within the destination. The messages continue until the image is either released or dragged out of the window or view. The return value should be one of the dragging operation options listed under the **draggingEntered:** method. The default return value (if this method isn't implemented by the destination) is the value returned by the previous **draggingEntered:** message.

This method provides the destination with an opportunity to modify the dragging operation depending on the position of the mouse pointer inside of the destination view or window object. For example, you may have several graphics or areas of text contained within the same view and wish to tailor the dragging

operation, or to ignore the drag event completely, depending upon which object is underneath the mouse pointer at the time when the user releases the dragged image and the **performDragOperation:** method is invoked.

You typically examine the contents of the pasteboard in the **draggingEntered:** method, where this examination is performed only once, rather than in the **draggingUpdated:** method, which is invoked multiple times.

Only one destination at a time receives a sequence of **draggingUpdated:** messages. If the mouse pointer is within the bounds of two overlapping views that are both valid destinations, the uppermost view receives these messages until the image is either released or dragged out.

See also: – **draggingExited:**, – **prepareForDragOperation:**

performDragOperation:

– (BOOL)**performDragOperation:**(id <NSDraggingInfo>)*sender*

Invoked after the released image has been removed from the screen and the previous **prepareForDragOperation:** message has returned YES. The destination should implement this method to do the real work of importing the pasteboard data represented by the image. If the destination accepts the data, it returns YES, otherwise it returns NO. The default is to return NO.

See also: – **concludeDragOperation:**

prepareForDragOperation:

– (BOOL)**prepareForDragOperation:**(id <NSDraggingInfo>)*sender*

Invoked when the image is released, if the most recent **draggingEntered:** or **draggingUpdated:** message returned an acceptable drag-operation value. Returns YES if the receiver agrees to perform the drag operation and NO if not.

See also: – **performDragOperation:**

NSDraggingInfo

Adopted By: no Rhapsody classes

Declared In: AppKit/NSDragging.h

Protocol Description

The NSDraggingInfo protocol declares methods that supply information about a *dragging session* (see the NSDraggingDestination protocol for definitions of dragging terms). NSDraggingInfo methods are designed to be invoked from within a class's implementation of NSDraggingDestination informal protocol methods. The Application Kit automatically passes an object that conforms to the NSDraggingInfo protocol as the argument to each of the methods defined by NSDraggingDestination. NSDraggingInfo messages should be sent to this object; you never need to create a class that implements the NSDraggingInfo protocol.

Method Types

Dragging-session information

– draggingSource
– draggingSourceOperationMask
– draggingDestinationWindow
– draggingPasteboard
– draggingSequenceNumber
– draggingLocation

Image information

– draggedImage
– draggedImageLocation

Sliding the image

– slideDraggedImageTo:

Instance Methods

draggedImage

– (NSImage *)**draggedImage**

Returns the image being dragged. This image object visually represents the data put on the pasteboard during the drag operation; however, it is the pasteboard data and not this image that are ultimately utilized in the dragging operation.

See also: – **draggedImageLocation**

draggedImageLocation

– (NSPoint)**draggedImageLocation**

Returns the current location of the dragged image's origin in the base coordinate system of the destination object's window. The image moves along with the mouse pointer (the position of which is given by **draggingLocation**) but may be positioned at some offset.

See also: – **draggedImage**

draggingDestinationWindow

– (NSWindow *)**draggingDestinationWindow**

Returns the destination window for the dragging operation. Either this window is the destination itself, or it contains the view object that is the destination.

draggingLocation

– (NSPoint)**draggingLocation**

Returns the current location of the mouse pointer in the base coordinate system of the destination object's window.

See also: – **draggedImageLocation**

draggingPasteboard

– (NSPasteboard *)**draggingPasteboard**

Returns the pasteboard object that holds the data being dragged. The dragging operation that is ultimately performed utilizes this pasteboard data and not the image returned by the **draggedImage** method.

draggingSequenceNumber

– (int)**draggingSequenceNumber**

Returns a number that uniquely identifies the dragging session.

draggingSource

– (id)**draggingSource**

Returns the source, or owner, of the dragged data or **nil** if the source isn't in the same application as the destination. The dragging source implements methods from the NSDraggingSource informal protocol.

draggingSourceOperationMask

– (unsigned int)**draggingSourceOperationMask**

Returns the dragging operation mask declared by the dragging source (through its **draggingSourceOperationMaskForLocal:** method). If the source permits dragging operations, the elements in the mask will be one or more of the following, combined using the C bitwise OR operator:

Option	Meaning
NSDragOperationCopy	The data represented by the image can be copied.
NSDragOperationLink	The data can be shared.
NSDragOperationGeneric	The operation can be defined by the destination.
NSDragOperationPrivate	The operation is negotiated privately between the source and the destination.
NSDragOperationAll	Combines all the above.

If the source does not permit any dragging operations, then method should return NSDragOperationNone.

If the user is holding down a modifier key during the dragging session and the source doesn't prohibit modifier keys from affecting the drag operation (through its **ignoreModifierKeysWhileDragging** method), then the operating system combines the dragging operation value that corresponds to the modifier key (see the descriptions below) with the source's mask using the C bitwise AND operator.

On Mach the modifier keys are associated with the dragging operation options shown below,

Modifier Key	Dragging Option
Control	NSDragOperationLink
Alternate	NSDragOperationCopy
Command	NSDragOperationGeneric

while on Windows the modifier keys are associated with the following dragging operation options.

Modifier Key	Dragging Option
Control	NSDragOperationCopy
Shift-Control	NSDragOperationLink
Alternate	NSDragOperationCopy

slideDraggedImageTo:

– (void)**slideDraggedImageTo:**(NSPoint)*aPoint*

Slides the image to *aPoint*, a specified location in the screen coordinate system. This method can be used to snap the image down to a particular location. It should only be invoked from within the destination's implementation of **prepareForDragOperation:**—in other words, after the user has released the image but before it's removed from the screen.

NSDraggingSource

(informal protocol)

Category Of: NSObject

Declared In: AppKit/NSDragging.h

Description

The NSDraggingSource informal protocol declares methods that are implemented by the source object in a dragging session (see the NSDraggingDestination protocol for definitions of dragging terms). The *dragging source* is specified as an argument to the **dragImage:at:offset:event:pasteboard:source: slideBack:** message, sent to a window or view object to initiate the dragging session.

Of the methods declared below, only **draggingSourceOperationMaskForLocal:** must be implemented. The other methods are invoked only if the dragging source implements them. All four methods are invoked automatically during a dragging session—you never send an NSDraggingSource message directly to an object.

Method Types

Specifying dragging options

– draggingSourceOperationMaskForLocal:
– ignoreModifierKeysWhileDragging

Responding to dragging sessions

– draggedImage:beganAt:
– draggedImage:endedAt:deposited:

Instance Methods

draggedImage:beganAt:

– (void)**draggedImage:**(NSImage *)*anImage* **beganAt:**(NSPoint)*aPoint*

Invoked when *anImage* is displayed but before it starts following the mouse. *aPoint* is the origin of the image in screen coordinates. This method provides the source object with an opportunity to respond to the initiation of a dragging session. For example, you might choose to have the source give a visual indication to the user that data is being dragged from the source.

See also: **– convertScreenToBase:** (NSWindow), **– convertBaseToScreen:** (NSWindow), **– convertPoint:fromView:** (NSView), **– convertPoint:toView:** (NSView)

draggedImage:endedAt:deposited:

– (void)**draggedImage:**(NSImage *)*anImage*
endedAt:(NSPoint)*aPoint*
deposited:(BOOL)*flag*

Invoked after *anImage* has been released and the dragging destination has been given a chance to operate on the data it represents. *aPoint* is the location of the image's origin in the screen coordinate system when it was released. A YES value for *flag* indicates that the destination accepted the dragged data, while a NO value indicates that it was rejected.

This method provides the source object with an opportunity to respond to either a successful or a failed dragging session. For example, if you are moving data from one location to another, you could use this method to make the source data disappear from its previous location, if the dragging session is successful, or reset itself to its previous state, in the event of a failure.

See also: **– convertScreenToBase:** (NSWindow), **– convertBaseToScreen:** (NSWindow), **– convertPoint:fromView:** (NSView), **– convertPoint:toView:** (NSView)

draggingSourceOperationMaskForLocal:

– (unsigned int)**draggingSourceOperationMaskForLocal:**(BOOL)*flag*

This is the only NSDraggingSource method that must be implemented by the source object. It should return a mask, built by combining the applicable constants listed below using the C bitwise OR operator. You should use this mask to indicate which types of dragging operations the source object will allow to be performed on the dragged image's data. A YES value for *flag* indicates that the candidate destination object (the window or view over which the dragged image is currently poised) is in the same application as the source, while a NO value indicates that the destination object is in a different application.

Option	Meaning
NSDragOperationCopy	The data represented by the image can be copied.
NSDragOperationLink	The data can be shared.
NSDragOperationGeneric	The operation can be defined by the destination.
NSDragOperationPrivate	The operation is negotiated privately between the source and the destination.
NSDragOperationAll	Combines all the above.

If the source does not permit any dragging operations, then it should return NSDragOperationNone.

ignoreModifierKeysWhileDragging

– (BOOL)**ignoreModifierKeysWhileDragging**

Sets whether the use of the modifier keys should have no effect on the type of operation performed. If this method is not implemented or returns NO, then the user can tailor the drag operation by holding down a modifier key during the drag. The dragging option that corresponds to the modifier key is combined with the source's mask (as set with the **draggingSourceOperationMaskForLocal:** method) using the C bitwise AND operator. See the description for the **draggingSourceOperationMask** method in the NSDraggingInfo protocol specification for more information about dragging masks and modifier keys.

NSIgnoreMisspelledWords

Adopted By: NSText

Declared In: AppKit/NSSpellProtocol.h

Protocol Description

Implement this protocol to have the Ignore button in the Spelling panel function properly. The Ignore button allows the user to accept a word that the spelling checker believes is misspelled. In order for this action to update the “ignored words” list for the document being checked, the NSIgnoreMisspelledWords protocol must be implemented.

This protocol is necessary because a list of ignored words is useful only if it pertains to the entire document being checked, but the spelling checker (NSSpellChecker object) does not check the entire document for spelling at once. The spelling checker returns as soon as it finds a misspelled word. Thus, it checks only a subset of the document at any one time. The user usually wants to check the entire document, so usually several spelling checks are run in succession until no misspelled words are found. This protocol allows the list of ignored words to be maintained per-document, even though the spelling checks are not run per-document.

The NSIgnoreMisspelledWords protocol specifies a method, **ignoreSpelling:**, which should be implemented like this:

```
- (void)ignoreSpelling:(id)sender
{
    [[NSSpellChecker sharedSpellChecker] ignoreWord:[[sender selectedCell]
        stringValue] inSpellDocumentWithTag:myDocumentTag];
}
```

The second argument to the NSSpellChecker method **ignoreWord:inSpellDocumentWithTag:** is a tag that the NSSpellChecker can use to distinguish the documents being checked. (See the discussion of “Matching a List of Ignored Words With the Document It Belongs To” in the description of the NSSpellChecker class.) Once the NSSpellChecker has a way to distinguish the various documents, it can append new ignored words to the appropriate list.

To make the ignored words feature useful, the application must store a document’s ignored words list with the document. See the NSSpellChecker class description for more information.

Instance Methods

ignoreSpelling:

– (void)**ignoreSpelling:**(id)*sender*

Implement to allow an application to ignore misspelled words on a document-by-document basis. This message is sent by the NSSpellChecker instance to the object whose text is being checked.

Implement this method by using the code shown in the protocol description.

NSMenuItem

Adopted By: NSMenuItem

Conforms To: NSCoding,
NSCopying
NSObject (NSObject)

Declared In: AppKit/NSMenuItem.h

Warning: The NSMenuItem protocol will be removed from the Application Kit in the Premier release of Rhapsody. The NSMenuItem class will solely assume all associated functionality. This change does not affect binary compatibility between different versions of projects, but might cause failures in project builds. To adapt your projects to this change, alter all references to the protocol (for example, “id <NSMenuItem>”) to references to the class (“NSMenuItem”).

Protocol Description

The NSMenuItem protocol declares methods that are used to manipulate command items in menus. The NSMenuItem class adopts this protocol, implementing all methods the protocol declares, and provides the basic functionality of command items. With some implementations of the OpenStep specification (including OPENSTEP), you cannot replace the NSMenuItem class with a different class which conforms to the NSMenuItem protocol. You may, however, subclass the NSMenuItem class if necessary.

The methods declared by the NSMenuItem protocol allow you to set the titles, actions, targets, tags, images, enabled states, and similar attributes of individual menu items, as well as to obtain the current values of these attributes. As implemented for the NSMenuItem class, a menu item, whenever one of its attributes changes, notifies the associated NSMenu via the **itemChanged:** method. The protocol also allows a conforming object to set keyboard equivalents and (for Microsoft Windows) mnemonics for menu items. See the sections below for more on this functionality.

See the NSMenu, NSMenuView, and NSMenuItemCell class specifications and the NSMenuValidation protocol specification for more information on menus.

Keyboard Equivalents

An object conforming to the NSMenuItem protocol can be assigned a keyboard eqivalent, so that when the user types a character the menu item’s action is sent. The keyboard eqivalent is defined in two parts. First is the basic key equivalent, which must be a Unicode character that can be generated by a single key press without modifier keys (Shift excepted). It is also possible to use a sequence of Unicode characters so long

as the user's key mapping is able to generate the sequence with a single key press. The basic key equivalent is set using **setKeyEquivalent:** and returned by **keyEquivalent**. The second part defines the modifier keys that must also be pressed. This is set using **setKeyEquivalentModifierMask:** and returned by **keyEquivalentModifierMask**. The modifier mask by default includes NSCommandKeyMask, and may also include the masks for the Shift, Alternate, or other modifier keys. Specifying keyboard equivalents in two parts allows you to define a modified keyboard equivalent without having to know which character is generated by the basic key plus the modifier. For example, you can define the keyboard equivalent Command-Alt-f without having to know which character is generated by typing Alt-f.

Certain methods in the NSMenuItem protocol can override assigned keyboard equivalents with those the user has specified in the defaults system. The **setUsesUserKeyEquivalents:** protocol method turns this behavior on or off, and **usesUserKeyEquivalents** returns its status. To determine the user-defined key equivalent for an NSMenuItem object, invoke the **userKeyEquivalent** instance method. If user-defined key equivalents are active and an NSMenuItem object has a user-defined key equivalent, its **keyEquivalent** method returns the user-defined key equivalent and not the one set using **setKeyEquivalent:**.

Mnemonics

On certain platforms, currently including Microsoft Windows, an object conforming to the NSMenuItem protocol can also be assigned a mnemonic. Mnemonics can be assigned on other platforms as well, however, they won't have any effect. Mnemonics are represented by an underlined character in the title of a menu item. The mnemonic can be any character that can be generated by a single key press without modifier keys (Shift excepted). When the menu is active, the user can type the underlined character in the menu item in order to activate that menu item. On Microsoft Windows a user activates the menu by pressing the Alternate key. A particular mnemonic character should only be used once within the set of menu items contained either in the same menu as the menu item or in the application's main menu.

Radio-Style Grouping

By using a few methods of the NSMenuItem protocol, you can implement radio-style groupings of menu commands. In other words, you can have a grouping of menu commands (usually segregated visually with separator items) and only one command in the group can be selected; the selected item is marked by an image, usually a radio-button image, but sometimes a checkmark. If the user selects another command in the group, the previous command is unmarked and the selected command displays the image. As an example of a radio-style grouping, a game could have three commands to indicate the level of play: Beginner, Intermediate, and Advanced.

To implement this feature, first set the images you want to use for the possible command states: "on," "off," and "mixed" (the last is useful for triple-state or indeterminate situations). To set the image, use the commands **setOnStateImage:**, **setOffStateImage:**, and **setMixedStateImage:**. The default image for the "on" state is a checkmark (NSMenuCheckmark) and for the "mixed" state the image is a dash (NSMenuMixedState). The "off" state typically has no image. The radio-button image (which you must set explicitly) is NSMenuRadio.

In an action method that responds to all commands in the group use **setState:** to uncheck the menu item that is currently marked:

```
[curItem setState:NSOffState];
```

Then mark the newly selected command:

```
[sender setState:NSOnState];
```

Method Types

Creating conforming NSMenuItem objects

– initWithTitle:action:keyEquivalent:

Enabling a menu item

– setEnabled:
– isEnabled

Setting the target and action

– setTarget:
– target
– setAction:
– action

Setting the title

– setTitle:
– title

Setting the tag

– setTag:
– tag

Setting the state

– setState:
– state

Setting the image

– setImage:
– image
– setOnStateImage:
– onStateImage
– setOffStateImage:
– offStateImage
– setMixedStateImage:
– mixedStateImage

Managing submenus

– setSubmenu:
– submenu
– hasSubmenu

Getting a separator item

+ separatorItem
– isSeparatorItem

Setting the owning menu

– setMenu:
– menu

Managing key equivalents

– setKeyEquivalent:
– keyEquivalent
– setKeyEquivalentModifierMask:
– keyEquivalentModifierMask

Managing mnemonics

– setMnemonicLocation:
– mnemonicLocation
– setTitleWithMnemonic:
– mnemonic

Managing user key equivalents

+ setUsesUserKeyEquivalents:
+ usesUserKeyEquivalents
– userKeyEquivalent

Representing an object

– setRepresentedObject:
– representedObject

Class Methods

separatorItem

+ (id <NSMenuItem>)**separatorItem**

Returns a menu item that is used to separate logical groups of menu commands. This menu item is disabled. The default separator item is a simple horizontal line.

See also: – **isSeparatorItem**, – **setEnabled:**

setUsesUserKeyEquivalents:

+ (void)**setUsesUserKeyEquivalents:**(BOOL)*flag*

If *flag* is YES, menu items conform to user preferences for key equivalents; otherwise, the key equivalents originally assigned to the menu items are used.

See also: **+ usesUserKeyEquivalents, – userKeyEquivalent**

usesUserKeyEquivalents

+ (BOOL)**usesUserKeyEquivalents**

Returns YES if menu items conform to user preferences for key equivalents; otherwise, returns NO.

See also: **+ setUsesUserKeyEquivalents:, – userKeyEquivalent**

Instance Methods

action

– (SEL)**action**

Returns the receiver's action method.

See also: **– target, – setAction:**

hasSubmenu

– (BOOL)**hasSubmenu**

Returns YES if the receiver has a submenu, NO if it doesn't.

See also: – **setSubmenu:forItem:**(NSMenu)

image

– (NSImage *)**image**

Returns the image displayed by the receiver, or **nil** if it displays no image.

See also: **– setImage:**

initWithTitle:action:keyEquivalent:

– (id)**initWithTitle:**(NSString *)*itemName*
action:(SEL)*anAction*
keyEquivalent:(NSString *)*charCode*

Returns an initialized instance of an object that conforms to the NSMenuItem protocol. The arguments *itemName* and *charCode* must not be **nil** (if there is no title or key equivalent, specify an empty NSString). The *anAction* argument must be a valid selector or NULL For instances of the NSMenItem class, the default initial state is NSStateOff, the default on-state image is a checkmark, and the default mixed-state image is a dash.

isEnabled

– (BOOL)**isEnabled**

Returns YES if the receiver is enabled, NO if not.

See also: – **setEnabled:**

isSeparatorItem

– (BOOL)**isSeparatorItem**

Returns whether the receiver is a separator item (that is, a menu item used to visually segregate related menu items).

See also: + **separatorItem**

keyEquivalent

– (NSString *)**keyEquivalent**

Returns the receiver's unmodified keyboard equivalent, or the empty string if one hasn't been defined. Use **keyEquivalentModifierMask** to determine the modifier mask for the key equivalent.

See also: **– userKeyEquivalent, – mnemonic, – setKeyEquivalent:**

keyEquivalentModifierMask

– (unsigned int)**keyEquivalentModifierMask**

Returns the receiver's keyboard equivalent modifier mask.

See also: **– setKeyEquivalentModifierMask:**

menu

– (NSMenu *)**menu**

Returns the menu to which the receiver belongs, or **nil** if no menu has been set.

See also: – **setMenu:**

mixedStateImage

– (NSImage *)**mixedStateImage**

Returns the image used to depict a “mixed state.” A mixed state is useful for indicating “off” and “on” attribute values in a group of selected objects, such as a selection of text containing bold and plain (non-bolded) worlds.

See also: – **setMixedStateImage:**

mnemonic

– (NSString *)mnemonic

Returns the character in the menu item title that appears underlined for use as a mnemonic. If there is no mnemonic character, returns an empty string.

See also: – **setTitleWithMnemonic:**

mnemonicLocation

– (unsigned int)**mnemonicLocation**

Returns the position of the underlined character in the menu item title used as a mnemonic. The position is the zero based index of that character in the title string. If the receiver has no mnemonic character, returns NSNotFound.

See also: – **setMnemonicLocation:**

offStateImage

– (NSImage *)offStateImage

Returns the image used to depict the receiver’s “off” state, or **nil** if the image has not been set.

See also: – **setOffStateImage:**

onStateImage

– (NSImage *)**onStateImage**

Returns the image used to depict the receiver's "on" state, or **nil** if the image has not been set.

See also: – **setOnStateImage:**

representedObject

– (id)**representedObject**

Returns the object that the receiving menu item represents. For example, you might have a menu list the names of views that are swapped into the same panel. The represented objects would be the appropriate NSView objects. The user would then be able to swtich back and forth between the different views that are displayed by selecting the various menu items.

See also: – **tag**, – **setRepresentedObject:**

setAction:

– (void)**setAction:**(SEL)*aSelector*

Sets the receiver's action method to *aSelector*.

See also: – **setTarget:**, – **action**

setEnabled:

– (void)**setEnabled:**(BOOL)*flag*

Sets whether the receiver is enabled based on *flag*. If a menu item is disabled, it's keyboard equivalent and mnemonic are also disabled. See the NSMenuValidation informal protocol specification for cautions regarding this method.

See also: – **isEnabled**

setImage:

– (void)**setImage:**(NSImage *)*menuImage*

Set's the receiver's image to *menuImage*. If *menuImage* is **nil**, the current image (if any) is removed. This image is not affected by changes in menu-item state.

See also: – **image**

setKeyEquivalent:

– (void)**setKeyEquivalent:**(NSString *)*aString*

Sets the receiver's unmodified key equivalent to *aString*. If you want to remove the key equivalent from a menu item, pass an empty string (@"") for *aString* (never pass **nil**). Use **setKeyEquivalentModifierMask:** to set the appropriate mask for the modifier keys for the key equivalent.

See also: **– setMnemonicLocation:, – keyEquivalent**

setKeyEquivalentModifierMask:

– (void)**setKeyEquivalentModifierMask:**(unsigned int)*mask*

Sets the receiver's keyboard equivalent modifiers (indicating modifiers such as the Shift or Alternate keys) to those in *mask*. *mask* is an integer bit field containing any of these modifier key masks, combined using the C bitwise OR operator:

NSShiftKeyMask
NSAlternateKeyMask
NSCommandKeyMask

On Mach, you should always set NSCommandKeyMask in *mask*; on Microsoft Windows, this is not required.

NSShiftKeyMask is relevant only for function keys; that is, for key events whose modifier flags include NSFunctionKeyMask. For all other key events NSShiftKeyMask is ignored and characters typed while the Shift key is pressed are interpreted as the shifted versions of those characters; for example, Command-Shift-'c' is interpreted as Command-'C'.

See the NSEvent class specification for more information about modifier mask values.

See also: **– keyEquivalentModifierMask**

setMenu:

– (void)**setMenu:**(NSMenu *)*aMenu*

Sets the receiver's menu to *aMenu*. This method is invoked by the owning NSMenu when the receiver is added or removed. You shouldn't have to invoke this method in your own code, although it can be overriden to provide specialized behavior.

See also: **– menu**

setMixedStateImage:

– (void)**setMixedStateImage:**(NSImage *)*itemImage*

Sets the image of the receiver that indicates a”mixed” state, that is, a state neither “on” or “off.” If *itemImage* is **nil**, any current mixed-state image is removed.

See also: – **mixedStateImage**, – **setOffStateImage:**, – **setOnStateImage:**, – **setState:**

setMnemonicLocation:

– (void)**setMnemonicLocation:**(unsigned int)*location*

Sets the character of the menu item title at *location* that is to be underlined. *location* must be between 0 and 254. This character identifies the access key on Windows by which users can access the menu item.

See also: – **mnemonicLocation**

setOffStateImage:

– (void)**setOffStateImage:**(NSImage *)*itemImage*

Sets the image of the receiver that indicates an “off” state. If *itemImage* is **nil**, any current off-state image is removed.

See also: – **offStateImage**, – **setMixedStateImage:**, – **setOnStateImage:**, – **setState:**

setOnStateImage:

– (void)**setOnStateImage:**(NSImage *)*itemImage*

Sets the image of the receiver that indicates an “on” state. If *itemImage* is **nil**, any current off-state image is removed.

See also: – **onStateImage**, – **setMixedStateImage:**, – **setOffStateImage:**, – **setState:**

setRepresentedObject:

– (void)**setRepresentedObject:**(id)*anObject*

Sets the object represented by the receiver to *anObject*. By setting a represented object for a menu item you make an association between the menu item and that object. The represented object functions as a more specific form of tag that allows you to associate any object, not just an **int**, with the items in a menu.

For example, an NSView object might be associated with a menu item—when the user chooses the menu item, the represented object is fetched and displayed in a panel. Several menu items might control the display of multiple views in the same panel.

See also: – **setTag:**, – **representedObject**

setState:

– (void)**setState:**(int)*itemState*

Sets the state of the receiver to *itemState*, which should be one of NSOffState, NSOnState, or NSMixedState. The image associated with the new state is displayed to the left of the menu item.

See also: – **state**, – **setMixedStateImage:**, – **setOffStateImage:**, – **setOnStateImage:**

setSubmenu:

– (void)**setSubmenu:**(NSMenu *)*aSubmenu*

Sets the submenu of the receiver to *aSubmenu*. The default implementation of the NSMenuItem class raises an exception if *aSubmenu* already has a supermenu.

See also: – **submenu**, – **hasSubmenu**

setTag:

– (void)**setTag:**(int)*anInt*

Sets the receiver's tag to *anInt*.

See also: – **setRepresentedObject:**, – **tag**

setTarget:

– (void)**setTarget:**(id)*anObject*

Sets the receiver's target to *anObject*.

See also: **– setAction:, – target**

setTitle:

– (void)**setTitle:**(NSString *)*aString*

Sets the receiver's title to *aString*.

See also: – **title**

setTitleWithMnemonic:

– (void)**setTitleWithMnemonic:**(NSString *)*aString*

Sets the title of a menu item with a character underlined to denote an access key (Windows only). Use an ampersand character to mark the character (the one following the ampersand) to be underlined. For example, the following message causes the 'c' in 'Receive' to be underlined:

```
[aMenuItem setTitleWithMnemonic:NSLocalizedString(@"Re&ceive")];
```

See also: – **mnemonic**, – **setMnemonicLocation:**

state

– (int)**state**

Returns the state of the receiver, which is NSOffState (the default), NSOnState, or NSMixedState.

See also: – **setState:**

submenu

– (NSMenu *)**submenu**

Returns the submenu associated with the receiving menu item, or **nil** if no submenu is associated with it. In the implementation of the NSMenuItem class, if the receiver responds YES to **hasSubmenu**, the submenu is returned.

See also: – **hasSubmenu**, – **setSubmenu**:

tag

– (int)**tag**

Returns the receiver's tag.

See also: – **representedObject**, – **setTag:**

target

– (id)**target**

Returns the receiver's target.

See also: **– action, – setTarget:**

title

– (NSString *)**title**

Returns the receiver's title.

See also: **– setTitle:**

userKeyEquivalent

– (NSString *)**userKeyEquivalent**

Returns the user-assigned key equivalent for the receiver.

See also: **– keyEquivalent**

NSMenuValidation

(informal protocol)

Category Of: NSObject

Declared In: AppKit/NSMenu.h

Protocol Description

This informal protocol allows your application to update the enabled or disabled status of an NSMenuItem. It declares only one method, **validateMenuItem:**. By default, every time a user event occurs, NSMenu automatically enables and disables each visible menu item based on criteria described in "Automatic Updating of NSMenuItems," below. Implement **validateMenuItem:** in cases where you want to override NSMenu's default enabling scheme.

NSMenuItems can be enabled or disabled in two ways: explicitly, by sending the **setEnabled:** message, or automatically, as described below. Automatic updating can be turned on and off with NSMenu's **setAutoenablesItems:** message.

Automatic Updating of NSMenuItems

Whenever a user event occurs, the NSMenu object updates the status of every one of its visible menu items. To update the status of a menu item, an NSMenu tries to find the object that responds to the NSMenuItem's action message. It searches the following objects in the given order until it finds one that responds to the action message (note that it doesn't actually send the action message):

- The NSMenuItem's target. If the target is non-**nil**, the search ends here whether the target responds or not.
- The key window's responder chain, starting with its first responder.
- The key window itself.
- The key window's delegate.
- The main window's responder chain, starting with its first responder.
- The main window itself.
- The main window's delegate.
- The NSApplication object.
- The NSApplication object's delegate.

If none of these objects responds to the action message, the menu item is disabled. If NSMenu finds an object that responds to the action message, it then checks to see if that object responds to the **validateMenuItem:** method (the method declared in this informal protocol). If **validateMenuItem:** is not

implemented in that object, the menu item is enabled. If it is implemented, the return value of **validateMenuItem:** indicates whether the menu item should be enabled or disabled.

Here is an example of using **validateMenuItem:** to override automatic enabling. If your application has a Copy menu item that sends the **copy:** action message to the first responder, that menu item is automatically enabled any time an object that responds to **copy:**, such as an NSText object, is the first responder of the key or main window. If you create a class whose instances might become the first responder, and which doesn't support copying of everything it allows the user to select, you should implement **validateMenuItem:** in that class. **validateMenuItem:** will then return NO if items that can't be copied are selected (or if no items are selected) and YES if all items in the selection can be copied. By implementing **validateMenuItem:**, you can have the Copy menu item disabled even though the target object does implement the **copy:** method. If a class never permits copying, then you simply omit an implemention of **copy:** in that class, and the Copy menu item is disabled automatically whenever an instance of that class is the first responder.

If you send a **setEnabled:** message to enable or disable a menu item when automatic updating is turned on (with NSMenu's **setAutoEnablesItems:**), other objects might undo what you have done after another user event occurs. Hence you can never be sure that the menu item will remain the way you set it. If your application must use **setEnabled:**, turn off the automatic enabling of menu items in order to get predictable results.

Instance Methods

validateMenuItem:

– (BOOL)**validateMenuItem:**(NSMenuItem *)*aMenuItem*

Implemented to override the default action of enabling or disabling *aMenuItem*. The object implementing this method must be the target of *aMenuItem*. It returns YES to enable the *aMenuItem*, NO to disable it. You can determine which menu item *aMenuItem* is by querying it for its title, tag, or action.

The following example beeps and disables the menu item "Next Record" if the selected line in a table view is the last one; conversely, it beeps and disables the menu item "Prior Record" if the selected row is the first one in the table view. (**countryKeys** is an array of names appearing in the table view.)

```
- (BOOL)validateMenuItem:(NSMenuItem *)anItem
{
    int row = [tableView selectedRow];
    if ([[anItem title] isEqualToString:@"Next Record"] &&
        (row == [countryKeys indexOfObject:[countryKeys lastObject]])) {
        return NO;
    }
    if ([[anItem title] isEqualToString:@"Prior Record"] && row == 0 ) {
        return NO;
    }
    return YES;
}
```

NSNibAwaking

(informal protocol)

Category Of: NSObject

Declared In: AppKit/NSNibLoading.h

Protocol Description

This informal protocol consists of a single method, **awakeFromNib**. Classes can implement this method to perform final initialization of state after objects have been loaded from an Interface Builder archive.

Instance Methods

awakeFromNib

– (void)**awakeFromNib**

Implemented to prepare the receiver for service after it has been loaded from an Interface Builder archive, or *nib file*. An **awakeFromNib** message is sent to each object loaded from the archive, but only if it can respond to the message, and only after all the objects in the archive have been loaded and initialized. When an object receives an **awakeFromNib** message, it's guaranteed to have all its outlet instance variables set.

Note: This method is also sent during Interface Builder's test mode to objects instantiated from loaded palettes, which include executable code for the objects. It isn't sent to objects defined solely by using the Classes display of the nib file window in Interface Builder.

When an Interface Builder archive is loaded into an application, each custom object from the archive is first initialized with an **init** message, or **initWithFrame:** if the object is a kind of NSView. It's then more specifically initialized with the properties that it was configured with using Interface Builder. This part of the initialization process uses any **set***Variable***:** methods that are available (where *variable* is the name of an instance variable whose value was set in Interface Builder). Finally, after all the objects are fully initialized, each receives an **awakeFromNib** message.

The order in which objects are loaded from the archive is not guaranteed. Therefore, it's possible for a **set***Variable***:** message to be sent to an object before its companion objects have been unarchived. For this reason, **set***Variable***:** methods should not send messages to other objects in the archive. However, messages to other objects can safely be sent from within **awakeFromNib**—by which time it's assured that all the objects are unarchived and initialized (though not necessarily awakened, of course).

Typically, **awakeFromNib** is implemented for classes whose instances are used as the owners of a loaded nib file (shown as "File's Owner" in Interface Builder). Such a class has the express purpose of connecting the loaded objects with objects in the application, and can thereafter be disposed of, or remain in the

capacity of a controller or coordinator for the loaded objects. For example, suppose that a nib file contains two custom views that must be positioned relative to each other at run time. Trying to position them when either one of the views is initialized (in **initWithCoder:** or a **set***Variable***:** method) might fail, since the other views might not be unarchived and initialized yet. However, it can be done in the nib file owner's **awakeFromNib** method (**firstView** and **secondView** are outlets of the file's owner):

```
- (void)awakeFromNib
{
    NSRect viewFrame;

    if ([[self superclass] instancesRespondToSelector:@selector(awakeFromNib)]) {
        [super awakeFromNib];
    }
    viewFrame = [firstView frame];
    viewFrame.origin.x += viewFrame.size.width;
    [secondView setFrame:viewFrame];
    return;
}
```

Note the testing of the superclass before invoking its implementation of **awakeFromNib**. The Application Kit declares a prototype for this method, but doesn't implement it. Because there's no default implementation of **awakeFromNib**, be sure to invoke it only when the object does in fact respond.

See also: + **loadNibNamed:owner:** (NSBundle Additions),
– **awakeAfterUsingCoder** (NSObject class of the Foundation Kit),
– **initWithCoder:** (NSCoding protocol of the Foundation Kit),
+ **initialize** (NSObject class of the Foundation Kit)

NSServicesRequests

(informal protocol)

Category Of: NSObject

Declared In: AppKit/NSApplication.h

Protocol Description

This informal protocol consists of two methods, **writeSelectionToPasteboard:types:** and **readSelectionFromPasteboard:**. The first is implemented to provide data to a remote service, and the second to receive any data the remote service might send back. Both respond to messages that are generated when the user chooses a command from the Services menu.

Instance Methods

readSelectionFromPasteboard:

- (BOOL)**readSelectionFromPasteboard:**(NSPasteboard *)*pboard*

Implemented to replace the current selection (that is, the text or objects that are currently selected) with data read from the Pasteboard object *pboard*. The data would have been placed in the pasteboard by another application in response to a remote message from the Services menu. A **readSelectionFromPasteboard:** message is sent to the same object that previously received a **writeSelectionToPasteboard:types:** message.

There's no default **readSelectionFromPasteboard:** method. The Application Kit declares a prototype for this method, but doesn't implement it.

writeSelectionToPasteboard:types:

- (BOOL)**writeSelectionToPasteboard:**(NSPasteboard *)*pboard* **types:**(NSArray *)*types*

Implemented to write the current selection to the Pasteboard object *pboard*. The selection should be written as one or more of the data types listed in *types*. After writing the data, this method should return YES. If for any reason it can't write the data, it should return NO.

A **writeSelectionToPasteboard:types:** message is sent to the first responder when the user chooses a command from the Services menu, but only if the receiver didn't return **nil** to a previous **validRequestorForSendType:returnType:** message.

After this method writes the data to the pasteboard, a remote message is sent to the application that provides the service the user requested. If the service provider supplies return data to replace the selection, the first responder will then receive a **readSelectionFromPasteboard:** message.

There's no default **writeSelectionToPasteboard:types:** method. The Application Kit declares a prototype for this method, but doesn't implement it.

See also: – **validRequestorForSendType:returnType:** (NSResponder class)

NSTableDataSource

(informal protocol)

Category Of: NSObject

Declared In: AppKit/NSTableView.h

Category Description

The NSTableDataSource category declares the methods that an NSTableView uses to access the contents of its data source object. It determines how many rows to display by sending a **numberOfRowsInTableView:** message, and accesses individual values with the **tableView: objectValueForTableColumn:row:** and **tableView:setObjectValue:forTableColumn:row:** methods. A data source must implement the first two methods to work with an NSTableView, but if it doesn't implement the third the NSTableView simply provides read-only access to its contents.

The NSTableView treats objects provided by its data source as values to be displayed in NSCell objects. If these objects aren't of common value classes—such as NSString, NSNumber, and so on—you'll need to create a custom NSFormatter to display them. See the NSFormatter class specification for more information.

Suppose that an NSTableView's column identifiers are set up as NSStrings containing the names of attributes for the column, such as "Last Name", "City", and so on, and that the data source stores its records as an NSMutableArray, called **records**, of NSMutableDictionary objects using those names as keys. Here's a small example, given as an ASCII property list:

```
(
    {
        "Last Name" = Anderson;
        "First Name" = James;
        Abode = apartment;
        City = "San Francisco";
    },
    {
        "Last Name" = Beresford;
        "First Name" = Keith;
        Abode = apartment;
        City = "Redwood City";
    }
)
```

With such a record structure, this implementation of **tableView:objectValueForTableColumn:row:** suffices to retrieve values for the NSTableView:

```
- (id)tableView:(NSTableView *)aTableView
    objectValueForTableColumn:(NSTableColumn *)aTableColumn
    row:(int)rowIndex
{
    id theRecord, theValue;

    NSParameterAssert(rowIndex >= 0 && rowIndex < [records count]);
    theRecord = [records objectAtIndex:rowIndex];
    theValue = [theRecord objectForKey:[aTableColumn identifier]];
    return theValue;
}
```

Here's the corresponding method for setting values:

```
- (void)tableView:(NSTableView *)aTableView
    setObjectValue:anObject
    forTableColumn:(NSTableColumn *)aTableColumn
    row:(int)rowIndex
{
    id theRecord;

    NSParameterAssert(rowIndex >= 0 && rowIndex < [records count]);
    theRecord = [records objectAtIndex:rowIndex];
    [theRecord setObject:anObject forKey:[aTableColumn identifier]];
    return;
}
```

Finally, **numberOfRowsInTableView:** simply returns the count of the NSArray:

```
- (int)numberOfRowsInTableView:(NSTableView *)aTableView
{
    return [records count];
}
```

In each case, the NSTableView that sends the message is provided as *aTableView*. A data source object that manages several sets of data can choose the appropriate set based on which NSTableView sends the message.

Method Types

Getting values

– numberOfRowsInTableView:
– tableView:objectValueForTableColumn:row:

Setting values

– tableView:setObjectValue:forTableColumn:row:

Instance Methods

numberOfRowsInTableView:

– (int)**numberOfRowsInTableView:**(NSTableView *)*aTableView*

Returns the number of records managed for *aTableView* by the data source object. An NSTableView uses this method to determine how many rows it should create and display.

tableView:objectValueForTableColumn:row:

– (id)**tableView:**(NSTableView *)*aTableView*
objectValueForTableColumn:(NSTableColumn *)*aTableColumn*
row:(int)*rowIndex*

Returns an attribute value for the record in *aTableView* at *rowIndex*. *aTableColumn* contains the identifier for the attribute, which you get by using NSTableColumn's **identifier** method. For example, if *aTableColumn* stands for the city that an employee lives in and *rowIndex* specifies the record for an employee who lives in Portland, this method returns an object with a string value of "Portland". See the category description for an example.

tableView:setObjectValue:forTableColumn:row:

– (void)**tableView:**(NSTableView *)*aTableView*
setObjectValue:(id)*anObject*
forTableColumn:(NSTableColumn *)*aTableColumn*
row:(int)*rowIndex*

Sets an attribute value for the record in *aTableView* at *rowIndex*. *anObject* is the new value, and *aTableColumn* contains the identifier for the attribute, which you get by using NSTableColumn's **identifier** method. See the category description for an example.

NSTextAttachmentCell

Adopted By: NSTextAttachmentCell

Declared In: AppKit/NSTextAttachment.h

Protocol Description

The NSTextAttachmentCell protocol declares the interface for objects that draw text attachment icons and handle mouse events on their icons. With the exceptions of **cellBaselineOffset:**, **setAttachment:** and **attachment**, all of these methods are implemented by the NSCell class and described in that class specification.

See the NSAttributedString and NSTextView class specifications for general information on text attachments.

Method Types

Drawing

– drawWithFrame:inView:
– highlight:withFrame:inView:

Cell size and position

– cellSize
– cellBaselineOffset

Event handling

– wantsToTrackMouse
– trackMouse:inRect:ofView:untilMouseUp:

Setting the attachment

– setAttachment:
– attachment

Instance Methods

attachment

– (NSTextAttachment *)**attachment**

Returns the text attachment object that owns the receiver.

See also: – **setAttachment:**

cellBaselineOffset

– (NSPoint)**cellBaselineOffset**

Returns the position where the attachment cell's image should be drawn in text, relative to the current point established in the glyph layout. The image should be drawn so that its lower left corner lies on this point.

See also: – **icon** (NSFileWrapper)

cellSize

– (NSSize)**cellSize**

Returns the size of the attachment's icon.

See also: – **icon** (NSFileWrapper), – **fileWrapper** (NSTextAttachment)

drawWithFrame:inView:

– (void)**drawWithFrame:**(NSRect)*cellFrame* **inView:**(NSView *)*aView*

Draws the receiver's image within *cellFrame* in *aView*, which should be the focus view.

See also: – **drawWithFrame:inView:** (NSCell), – **lockFocus** (NSView)

highlight:withFrame:inView:

– (void)**highlight:**(BOOL)*flag*
withFrame:(NSRect)*cellFrame*
inView:(NSView *)*aView*

Draws the receiver's image—with highlighting if *flag* is YES—within *cellFrame* in *aView*, which should be the focus view.

See also: – **highlight:withFrame:inView:** (NSCell), – **lockFocus** (NSView)

setAttachment:

– (void)**setAttachment:**(NSTextAttachment *)*anAttachment*

Sets the text attachment object that owns the receiver to *anAttachment*, without retaining it (the text attachment, as the owner, retains the cell).

See also: – **attachment**, – **setAttachmentCell:** (NSTextAttachment)

trackMouse:inRect:ofView:untilMouseUp:

– (BOOL)**trackMouse:**(NSEvent *)*theEvent*
inRect:(NSRect)*cellFrame*
ofView:(NSView *)*aTextView*
untilMouseUp:(BOOL)*flag*

Handles a mouse-down event on the receiver's image. *theEvent* is the mouse-down event. *cellFrame* is the region of *aTextView* in which further mouse events should be tracked. *aTextView* is the view which received the event. It's assumed to be an NSTextView, and should be the focus view. If *flag* is YES, the receiver tracks the mouse until a mouse-up event occurs; if *flag* is NO, it stops tracking when a mouse-dragged event occurs outside of *cellFrame*. Returns YES if the receiver successfully finished tracking the mouse (typically through a mouse-up event), NO otherwise (such as when the mouse is dragged outside *cellFrame*).

NSTextAttachmentCell's implementation of this method calls upon *aTextView*'s delegate to handle the event. If *theEvent* is a mouse-up event for a double click, the text attachment cell sends the delegate a **textView:doubleClickedOnCell:inRect:** message and returns YES. Otherwise, depending on whether the user clicks or drags the cell, it sends the delegate a **textView:clickedOnCell:inRect:** or a **textView: draggingCell:inRect:event:** message and returns YES. NSTextAttachmentCell's implementation returns NO only if *flag* is NO and the mouse is dragged outside of *cellFrame*. The delegate methods are invoked only if the delegate responds.

See also: – **wantsToTrackMouse**, – **trackMouse:inRect:ofView:untilMouseUp:** (NSCell),
– **lockFocus** (NSView)

wantsToTrackMouse

– (BOOL)**wantsToTrackMouse**

Returns YES if the receiver will handle a mouse event occurring over its image (to support dragging, for example), NO otherwise. NSTextAttachmentCell's implementation of this method returns YES. The NSView containing the cell should invoke this method before sending a **trackMouse:inRect:ofView: untilMouseUp:** message.

For an attachment in an attributed string, if the attachment cell returns NO its attachment character should be selected rather than the cell being asked to track the mouse. this results in the attachment icon behaving as any regular glyph in text.

NSTextInput

Adopted By: NSInputManager

Declared In: AppKit/NSInputManager.h

Protocol Description

Note: *This class specification is incomplete and has not received a technical review. It is included in this release to test the linkage between the application development tools and the on-line documentation. What information it contains should be considered preliminary and subject to change.*

The methods of the NSTextInput protocol are implemented by objects—for example, responders—that handle text input. A client object such as a text view ("the client object") that speaks this protocol must be reponsible for the following things:

1. The client object must maintain a "marked region" within which text input and, possibly, character conversion take place. The marked region may have a length of zero. The client object must maintain an "insertion point"—typically at the end of the marked region, though it may be within the region. The "selection" within the client object, if any, is entirely contained within the marked region whenever there is a marked region.2. The client object is responsible for sending messages to currentInputManager when the mouse goes down inside the marked region, or when the mouse leaves the marked region. Within the marked region, this allows the selection to be changed. Out of the marked region, it allows the region to be "abandoned". (See below.)

3. When there is a non-zero marked region, the client object is responsible for notifying the input manager when the selection changes, or when other programmatic changes to the text affect the marked region. It can do this by sending a message to set the selection, abandon the marked region, etc.

4. When the client object relinquishes first responder, it will typically send markedTextWillBeAbandoned: to the currentInputManager. It must send markedTextWillBeAbandoned: when its insertion point (or selection moves outside the marked range. The server will typically respond by simply unmarking the region, but may remove the marked region's text entirely.

Note: If this protocol is not implemented by a client object that does have a keyDown: method, then in-line input is not possible for that client object, and will have to be handled externally.

The NSTextInput protocol is implemented by a IM to receive input from the server on behalf of the current client, and otherwise mediate between the client object and the server. It then forwards the corresponding messages to the client, or gets information from the client to pass back to the server, as appropriate.

The message passing between NSApp, IM, UIobj, and Server is all synchronous. That is, e.g., when IM sends a message to Server, any reply comes back and is relayed to UIobj before the original message returns.

A key binding manager splits the stream of keyDown: messages (intercepted by NSResopnder) into commands and text. If there is an Input Manager in the loop, it will further use any of these NSTextInput messages to control the marked region.

Method Types

Marking text

– setMarkedText:selectedRange:
– getMarkedText:selectedRange:
– hasMarkedText
– unmarkText

Other

– conversationIdentifier
– doCommandBySelector:
– insertText:

Instance Methods

conversationIdentifier

– (long)**conversationIdentifier**

Returns a number used to identify the receiver's input management session to the input server.

<<more information forthcoming>>

doCommandBySelector:

– (void)**doCommandBySelector:**(SEL)*aSelector*

Attempts to invoke *aSelector* or pass the message up the responer chain. This method is invoked by an input manager in response to an **interpretKeyEvents:** message.

<<more information forthcoming>>

See also: – **interpretKeyEvents:** (NSResponder)

getMarkedText:selectedRange:

– (void)**getMarkedText:**(out NSString **)*aString* **selectedRange:**(out NSRange *)*aRange*

Returns by reference in *aString* the receiver's marked text, if any, and in *aRange* the range of the selection within *aString* (not in terms of the receiver's entire text stream).

hasMarkedText

– (BOOL)**hasMarkedText**

Returns YES if the receiver has text that's still being interpreted by the input manager, NO if it doesn't.

<<more information forthcoming>>

insertText:

– (void)**insertText:**(NSString *)*aString*

Inserts *aString* into the receiver's text stream. This method is invoked by an input manager in response to an **interpretKeyEvents:** message.

<<more information forthcoming>>

See also: – **interpretKeyEvents:** (NSResponder)

setMarkedText:selectedRange:

– (void)**setMarkedText:**(NSString *)*aString* **selectedRange:**(NSRange)*selRange*

<<forthcoming>>

unmarkText

– (void)**unmarkText**

Removes any marking from pending input text, and accepts the text in its current state.

<<more information forthcoming>>

Defined Types

NSBorderType

DECLARED IN AppKit/NSView.h

SYNOPSIS

typedef enum _NSBorderType {
    NSNoBorder,
    NSLineBorder,
    NSBezelBorder,
    NSGrooveBorder
} **NSBorderType**;

DESCRIPTION This type represents the kinds of border that can be drawn around certain NSView subclasses.

NSButtonType

DECLARED IN AppKit/NSButtonCell.h

SYNOPSIS

typedef enum _NSButtonType {
    NSMomentaryPushButton,
    NSPushOnPushOffButton,
    NSToggleButton,
    NSSwitchButton,
    NSRadioButton,
    NSMomentaryChangeButton,
    NSOnOffButton,
    NSMomentaryLight,
} **NSButtonType**;

DESCRIPTION This type represents the way NSButtons and NSButtonCells behave when pressed, and the way they display their state. See NSButton's and NSButtonCell's **setButtonType:** methods for more information.

NSCellAttribute

DECLARED IN AppKit/NSCell.h

SYNOPSIS

```
typedef enum _NSCellAttribute {
    NSCellDisabled,
    NSCellState,
    NSPushInCell,
    NSCellEditable,
    NSChangeGrayCell,
    NSCellHighlighted,
    NSCellLightsByContents,
    NSCellLightsByGray,
    NSChangeBackgroundCell,
    NSCellLightsByBackground,
    NSCellIsBordered,
    NSCellHasOverlappingImage,
    NSCellHasImageHorizontal,
    NSCellHasImageOnLeftOrBottom,
    NSCellChangesContents,
    NSCellIsInsetButton,
    NSCellAllowsMixedState
} NSCellAttribute;
```

DESCRIPTION This is the type of the first argument to the NSCell methods **setCellAttribute:to:** and **cellAttribute:** methods. Some of the values apply not to NSCell but to one of its subclasses.

Often it's preferable to change cell attributes using more specialized methods like **setState:** or **setEditable:**.

NSCellImagePosition

DECLARED IN AppKit/NSCell.h

SYNOPSIS

typedef enum _NSCellImagePosition {
    NSNoImage,
    NSImageOnly,
    NSImageLeft,
    NSImageRight,
    NSImageBelow,
    NSImageAbove,
    NSImageOverlaps,
} **NSCellImagePosition**;

DESCRIPTION These constants represent the position of an NSButtonCell's NSImage relative to its title. See NSButton's and NSButtonCell's **setImagePosition:** and **imagePosition** methods for more information.

NSCellType

DECLARED IN AppKit/NSCell.h

SYNOPSIS

typedef enum _NSCellType {
    NSNullCellType,
    NSTextCellType,
    NSImageCellType,
} **NSCellType**;

DESCRIPTION The values of this type determine what kind of data an NSCell displays. NSCells of type NSTextCellType and NSImageCellType display text and images, respectively. NSCells of type NSNullCellType display nothing. See the NSCell methods **type** and **setType:** for more information.

NSDataLinkDisposition

DECLARED IN AppKit/NSDataLink.h

SYNOPSIS

```
typedef enum _NSDataLinkDisposition {
    NSLinkInDestination,
    NSLinkInSource,
    NSLinkBroken
} NSDataLinkDisposition;
```

DESCRIPTION Returned by NSDataLink's **disposition** method to identify a link as a destination link, a source link, or a broken link. See the NSDataLink class specification for more information on the dispositions of links.

NSDataLinkNumber

DECLARED IN AppKit/NSDataLink.h

SYNOPSIS

```
typedef int NSDataLinkNumber;
```

DESCRIPTION The type returned by NSDataLink's **linkNumber** method as a persistent identifier of a destination link.

NSDataLinkUpdateMode

DECLARED IN AppKit/NSDataLink.h

SYNOPSIS

```
typedef enum _NSDataLinkUpdateMode {
    NSUpdateContinuously,
    NSUpdateWhenSourceSaved,
    NSUpdateManually,
    NSUpdateNever
} NSDataLinkUpdateMode;
```

DESCRIPTION Used by NSDataLink's **setUpdateMode:** and **updateMode** methods to identify when a link's data is to be updated.

NSEventType

DECLARED IN AppKit/NSEvent.h

SYNOPSIS

```
typedef enum _NSEventType {
        NSLeftMouseDown,
        NSLeftMouseUp,
        NSRightMouseDown,
        NSRightMouseUp,
        NSMouseMoved,
        NSLeftMouseDragged,
        NSRightMouseDragged,
        NSMouseEntered,
        NSMouseExited,
        NSKeyDown,
        NSKeyUp,
        NSFlagsChanged,
        NSAppKitDefined,
        NSSystemDefined,
        NSApplicationDefined
        NSPeriodic,
        NSCursorUpdate
} NSEventType;
```

DESCRIPTION This type represents various kinds of events. It is the return type of NSEvent's type method, and the type of the first argument to NSEvent's **...EventWithType:** methods.

NSFontAction

DECLARED IN AppKit/NSFontManager.h

SYNOPSIS

typedef enum _NSFontAction {
    NSNoFontChangeAction,
    NSViaPanelFontAction,
    NSAddTraitFontAction,
    NSSizeUpFontAction,
    NSSizeDownFontAction,
    NSHeavierFontAction,
    NSLighterFontAction,
    NSRemoveTraitFontAction
} **NSFontAction**;

DESCRIPTION Values of this type tag the actions of font menu cells. When a font menu cell sends a message to NSFontManager, NSFontManager checks the cell for one of these tags.

This type is in the API for explanatory purposes only. You will never use it directly.

NSFontTraitMask

DECLARED IN AppKit/NSFontManager.h

SYNOPSIS typedef unsigned int **NSFontTraitMask;**

DESCRIPTION An NSFontTraitMask characterizes one or more of a font's traits. It's used as an argument type for NSAttributedString's **applyFontTraits:range:** method, NSCStringText's **setSelFontStyle:** method, and several of the methods in the NSFontManager class.

NSGlyph

DECLARED IN AppKit/NSFont.h

SYNOPSIS typedef unsigned int **NSGlyph**;

DESCRIPTION This type is used to specify PostScript glyphs in such NSFont methods as **glyphWithName:**.

NSGlyphInscription

DECLARED IN AppKit/NSLayoutManager.h

SYNOPSIS

```
typedef enum {
    NSGlyphInscribeBase,
    NSGlyphInscribeBelow,
    NSGlyphInscribeAbove,
    NSGlyphInscribeOverstrike,
    NSGlyphInscribeOverBelow
} NSGlyphInscription;
```

DESCRIPTION The inscribe attribute of an glyph determines how it is laid out relative to the previous glyph.

NSGlyphRelation

DECLARED IN AppKit/NSFont.h

SYNOPSIS

```
typedef enum _NSGlyphRelation {
    NSGlyphBelow,
    NSGlyphAbove,
} NSGlyphRelation;
```

DESCRIPTION This type specifies the position of a glyph in relation to the base glyph. Parameters of this type are used in the second slot of the NSFont method **positionOfGlyph:withRelation: toBaseGlyph:...**.

NSGradientType

DECLARED IN AppKit/NSButtonCell.h

SYNOPSIS

typedef enum _NSGradientType {
    NSGradientNone,
    NSGradientConcaveWeak,
    NSGradientConcaveStrong,
    NSGradientConvexWeak,
    NSGradientConvexStrong
} **NSGradientType**;

DESCRIPTION This type represents the darkness gradient of an NSButtonCell. A concave gradient is darkest in the top left corner, a convex gradient is darkest in the bottom right corner. A weak gradient has only weak darkness contrast between opposite corners; a strong gradient has strong contrast. See the NSButtonCell methods **gradient** and **setGradient:** for more information.

NSImageAlignment

DECLARED IN AppKit/NSImageCell.h

SYNOPSIS

typedef enum {
    NSImageAlignCenter,
    NSImageAlignTop,
    NSImageAlignTopLeft,
    NSImageAlignTopRight,
    NSImageAlignLeft,
    NSImageAlignBottom,
    NSImageAlignBottomLeft,
    NSImageAlignBottomRight,
    NSImageAlignRight
} **NSImageAlignment**;

DESCRIPTION This type defines the ways of aligning an NSImage within an NSImageCell. It is the return type for NSImageCell's and NSImageView's **imageAlignment** methods, and an argument type for their **setImageAlignment:** methods.

NSImageFrameStyle

DECLARED IN AppKit/NSImageCell.h

SYNOPSIS

```
typedef enum {
    NSImageFrameNone,
    NSImageFramePhoto,
    NSImageFrameGrayBezel,
    NSImageFrameGroove,
    NSImageFrameButton
} NSImageFrameStyle;
```

DESCRIPTION This type defines the kinds of frames that can appear around an NSImageCell. It is the return type for NSImageCell's and NSImageView's **imageFrameStyle** methods, and an argument type for their **setImageFrameStyle:** methods.

NSImageScaling

DECLARED IN AppKit/NSImageCell.h

SYNOPSIS

```
typedef enum {
    NSScaleProportionally,
    NSScaleToFit,
    NSScaleNone
} NSImageScaling;
```

DESCRIPTION This type defines the ways that an image can be scaled to fit an NSImageCell. The value NSScaleProportionally means that the image should be scaled in a way that preserves its proportions. The value NSScaleToFit means that the image should fit the NSView, even if that means its proportions must be distorted. The value NSScaleNone means that the image's size should be preserved, even if it must be clipped to fit the NSView.

NSInterfaceStyle

DECLARED IN AppKit/NSInterfaceStyle.h

SYNOPSIS

```
typedef enum {
    NSNoInterfaceStyle,
    NSNextStepInterfaceStyle,
    NSWindows95InterfaceStyle,
    NSMacintoshInterfaceStyle
} NSInterfaceStyle;
```

DECLARED IN This type defines the style of an application's user interface. It is returned by the **interfaceStyle** method and taken as an argument by the **setInterfaceStyle:** method. Both of the these methods are in the NSInterfaceStyle category of NSResponder.

For more information, see the function **NSInterfaceStyleForKey**.

NSLineBreakMode

DECLARED IN AppKit/NSParagraphStyle.h

SYNOPSIS

```
typedef enum _NSLineBreakMode {
    NSLineBreakByWordWrapping,
    NSLineBreakByCharWrapping,
    NSLineBreakByClipping,
    NSLineBreakByTruncatingHead,
    NSLineBreakByTruncatingTail,
    NSLineBreakByTruncatingMiddle
} NSLineBreakMode;
```

DESCRIPTION This type defines the ways that a long paragraph can be broken into lines. The possible values are described below.

Value	Meaning
NSLineBreakByWordWrapping	The default value. At the last possible word boundary, the paragraph wraps to the next line.

Value	Meaning
NSLineBreakByCharWrapping	At the last possible character, the paragraph wraps to the next line.
NSLineBreakByClipping	As much of the paragraph appears as will fit on a single line. This value has the same effect as NSLineBreakByTruncatingTail
NSLineBreakByTruncatingHead	As much of the paragraph appears as will fit on a single line. Characters from the start of the paragraph do not appear.
NSLineBreakByTruncatingTail	As much of the paragraph appears as will fit on a single line. Characters from the end of the paragraph do not appear.
NSLineBreakByTruncatingMiddle	As much of the paragraph appears as will fit on a single line. Characters from the middle of the paragraph do not appear.

NSLineMovementDirection

DECLARED IN AppKit/NSTextContainer.h

SYNOPSIS

```
typedef enum {
    NSLineDoesntMove,
    NSLineMovesLeft,
    NSLineMovesRight,
    NSLineMovesDown,
    NSLineMovesUp
} NSLineMovementDirection;
```

DESCRIPTION This is an argument type for the NSTextContainer method **lineFragmentRectForProposedRect: sweepDirection:movementDirection:remainingRect:**.

NSLineSweepDirection

DECLARED IN AppKit/NSTextContainer.h

SYNOPSIS

typedef enum {
    NSLineSweepLeft,
    NSLineSweepRight,
    NSLineSweepDown,
    NSLineSweepUp
} **NSLineSweepDirection**;

DESCRIPTION This is an argument type for the NSTextContainer method **lineFragmentRectForProposedRect: sweepDirection:movementDirection:remainingRect:**.

NSMatrixMode

DECLARED IN AppKit/NSMatrix.h

SYNOPSIS

typedef enum _NSMatrixMode {
    NSRadioModeMatrix,
    NSHighlightModeMatrix,
    NSListModeMatrix,
    NSTrackModeMatrix
} **NSMatrixMode**;

DESCRIPTION These constants represent the modes of operation of an NSMatrix, as described in the NSMatrix class specification.

NSModalSession

DECLARED IN AppKit/NSApplication.h

SYNOPSIS

typedef struct _NSModalSession ***NSModalSession**;

DESCRIPTION Variables of type NSModalSession point to information used by the system between **beginModalSession:for:** and **endModalSession:** messages.

NSPrinterTableStatus

DECLARED IN AppKit/NSPrinter.h

SYNOPSIS

typedef enum _NSPrinterTableStatus {
    NSPrinterTableOK,
    NSPrinterTableNotFound,
    NSPrinterTableError
} **NSPrinterTableStatus**;

DESCRIPTION These constants are used to describe the state of a printer-information table stored by an NSPrinter object. See the NSPrinter method **statusForTable:** for more information.

NSPrintingOrientation

DECLARED IN AppKit/NSPrintInfo.h

SYNOPSIS

typedef enum _NSPrintingOrientation {
    NSPortraitOrientation,
    NSLandscapeOrientation
} **NSPrintingOrientation**;

DESCRIPTION These constants represent the way a page is oriented for printing. In NSPortraitOrientation, the page is taller than it is wide; in NSLandscapeOrientation, the page is wider than it is tall. See the NSPrintInfo methods **orientation** and **setOrientation:** for more information.

NSPrintingPageOrder

DECLARED IN AppKit/NSPrintOperation.h

SYNOPSIS

typedef enum _NSPrintingPageOrder {
    NSDescendingPageOrder,
    NSSpecialPageOrder,
    NSAscendingPageOrder,
    NSUnknownPageOrder
} **NSPrintingPageOrder**;

DESCRIPTION This type represents the order in which pages are to be printed. The value NSSpecialPageOrder tells the spooler to not rearrange the pages. The value NSUnknownPageOrder means that no page order is written out. See the NSPrintOperation methods **pageOrder** and **setPageOrder:** for more information.

NSPrintingPaginationMode

DECLARED IN AppKit/NSPrintInfo.h

SYNOPSIS

typedef enum _NSPrintingPaginationMode {
    NSAutoPagination,
    NSFitPagination,
    NSClipPagination
} **NSPrintingPaginationMode**;

DESCRIPTION These constants represent the different ways in which an image is divided into pages. The value NSFitPagination forces the image to fit on one page. The value NSClipPagination allows the image to be clipped by the page.See the NSPrintInfo class specification for a fuller explanation.

NSRulerOrientation

DECLARED IN AppKit/NSRulerView.h

SYNOPSIS

```
typedef enum {
    NSHorizontalRuler,
    NSVerticalRuler
} NSRulerOrientation;
```

DESCRIPTION This type defines whether an NSRulerView will be displayed horizontally or vertically. It is the return type of NSRulerView's **orientation** method, and an argument to the NSRulerView methods **setOrientation:** and **initWithScrollView:orientation:**.

NSScrollArrowPosition

DECLARED IN AppKit/NSScroller.h

SYNOPSIS

```
typedef enum _NSScrollArrowPosition {
    NSScrollerArrowsMaxEnd,
    NSScrollerArrowsMinEnd,
    NSScrollerArrowsNone
} NSScrollArrowPosition;
```

DESCRIPTION These constants are used in NSScroller's **setArrowsPosition:** method to set the position of the arrows within the scroller.

NSScrollerArrow

DECLARED IN AppKit/NSScroller.h

SYNOPSIS

```
typedef enum _NSScrollerArrow {
    NSScrollerIncrementArrow,
    NSScrollerDecrementArrow
} NSScrollerArrow;
```

DESCRIPTION This is the type of the first argument to the NSScroller method **drawArrow:highlight:**. The value determines which scroll button is drawn.

NSScrollerPart

DECLARED IN AppKit/NSScroller.h

SYNOPSIS

```
typedef enum _NSScrollerPart {
    NSScrollerNoPart,
    NSScrollerDecrementPage,
    NSScrollerKnob,
    NSScrollerIncrementPage,
    NSScrollerDecrementLine,
    NSScrollerIncrementLine,
    NSScrollerKnobSlot
} NSScrollerPart;
```

DESCRIPTION These constants are used in Scroller's **hitPart** method to identify the part of the Scroller specified in a mouse event.

NSSelectionAffinity

DECLARED IN AppKit/NSTextView.h

SYNOPSIS

```
typedef enum _NSSelectionAffinity {
    NSSelectionAffinityUpstream,
    NSSelectionAffinityDownstream
} NSSelectionAffinity;
```

DESCRIPTION This is the return type of the NSTextView method **selectionAffinity**, and the type of the second argument to the NSTextView method **setSelectedRange:affinity:stillSelecting:**.

NSSelectionDirection

DECLARED IN AppKit/NSWindow.h

SYNOPSIS

typedef enum _NSSelectionDirection {
    NSDirectSelection,
    NSSelectingNext,
    NSSelectingPrevious
} **NSSelectionDirection**;

DESCRIPTION This is the return type of the NSWindow method **keyViewSelectionDirection**.

NSSelectionGranularity

DECLARED IN AppKit/NSTextView.h

SYNOPSIS

typedef enum _NSSelectionGranularity {
    NSSelectByCharacter,
    NSSelectByWord,
    NSSelectByParagraph
} **NSSelectionGranularity**;

DESCRIPTION This is the return type of the NSTextView method selectionGranularity, and the type of arguments to two other NSTextView methods, **setSelectionGranularity:** and **selectionRangeForProposedRange:granularity:**.

NSTextAlignment

DECLARED IN AppKit/NSText.h

SYNOPSIS

typedef enum _NSTextAlignment {
    NSLeftTextAlignment,
    NSRightTextAlignment,
    NSCenterTextAlignment,
    NSJustifiedTextAlignment,
    NSNaturalTextAlignment
} **NSTextAlignment**;

DESCRIPTION Variables of this type are used as arguments and return values for methods that specify text alignment.

NSTextTabType

DECLARED IN AppKit/NSParagraphStyle.h

SYNOPSIS

typedef enum _NSTextTabType {
    NSLeftTabStopType,
    NSRightTabStopType,
    NSCenterTabStopType,
    NSDecimalTabStopType
} **NSTextTabType**;

DESCRIPTION This is the return type of NSTextTab's **tabStopType** method, and an argument to NSTextTab's **initWithType:location:** method.

NSTIFFCompression

SYNOPSIS AppKit/NSBitmapImageRep.h

SYNOPSIS

typedef enum _NSTIFFCompression {
NSTIFFCompressionNone,
NSTIFFCompressionCCITTFAX3,
NSTIFFCompressionCCITTFAX4,
NSTIFFCompressionLZW,
NSTIFFCompressionJPEG,
NSTIFFCompressionNEXT,
NSTIFFCompressionPackBits,
NSTIFFCompressionOldJPEG
} **NSTIFFCompression**;

DESCRIPTION

These constants represent the various TIFF (*tag image file format*) data compression schemes. See the NSBitmapImageRep class specification for their meanings.

NSTitlePosition

DECLARED IN

AppKit/NSBox.h

SYNOPSIS

typedef enum _NSTitlePosition {
NSNoTitle,
NSAboveTop,
NSAtTop,
NSBelowTop,
NSAboveBottom,
NSAtBottom,
NSBelowBottom
} **NSTitlePosition**;

DESCRIPTION

This type represents the locations where an NSBox's title can be placed with respect to its border. Thus, for example, NSAboveTop means the title is above the top of the border, NSAtTop means the title breaks the top border, and so on. See the NSBox methods **titlePosition** and **setTitlePosition:**.

NSTrackingRectTag

DECLARED IN AppKit/NSView.h

SYNOPSIS typedef int **NSTrackingRectTag**;

DESCRIPTION This type describes the rectangle used to track the mouse. See the NSView methods **addTrackingRect:...** and **removeTrackingRect:**.

NSUsableScrollerParts

DECLARED IN AppKit/NSScroller.h

SYNOPSIS

```
typedef enum _NSUsableScrollerParts {
    NSNoScrollerParts,
    NSOnlyScrollerArrows,
    NSAllScrollerParts
} NSUsableScrollerParts;
```

DESCRIPTION This type defines the usable parts of an NSScroller; see the class specification for more information.

NSWindowDepth

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS typedef int **NSWindowDepth**;

DESCRIPTION This type represents the depth, or amount of memory, devoted to a single pixel in a window or screen.

Enumerations

NSApplication—Modal Session Return Values

DECLARED IN AppKit/NSApplication.h

SYNOPSIS

```
enum {
    NSRunStoppedResponse,
    NSRunAbortedResponse,
    NSRunContinuesResponse
};
```

DESCRIPTION Return values for **the NSApplication methods runModalFor:** and **runModalSession:**.

NSAttributedString—Underlining

DECLARED IN AppKit/NSAttributedString.h

SYNOPSIS

```
enum {
    NSSingleUnderlineStyle
};
```

DESCRIPTION This defines the only currently supported value for NSUnderlineStyleAttributeName.

NSButtonCell—State Masks

DECLARED IN AppKit/NSCell.h

SYNOPSIS

```
enum {
    NSNoCellMask,
    NSContentsCellMask,
    NSPushInCellMask,
    NSChangeGrayCellMask,
    NSChangeBackgroundCellMask
};
```

DESCRIPTION These masks are passed to the NSButtonCell methods **highlightsBy:** and **showsStateBy:**.

NSCell—Action Flags

DECLARED IN AppKit/NSEvent.h

SYNOPSIS

```
enum {
    NSLeftMouseDownMask,
    NSLeftMouseUpMask,
    NSRightMouseDownMask,
    NSRightMouseUpMask,
    NSMouseMovedMask,
    NSLeftMouseDraggedMask,
    NSRightMouseDraggedMask,
    NSMouseEnteredMask,
    NSMouseExitedMask,
    NSKeyDownMask,
    NSKeyUpMask,
    NSFlagsChangedMask,
    NSAppKitDefinedMask,
    NSSystemDefinedMask,
    NSApplicationDefinedMask,
    NSPeriodicMask,
    NSCursorUpdateMask,
    NSAnyEventMask
};
```

DESCRIPTION These constants are masks for different kinds of events. You pass them to NSCell's **sendActionOn:** method to indicate when an NSCell should send its action message.

NSCell—Data Entry Types

DECLARED IN AppKit/NSCell.h

SYNOPSIS

```
enum {
    NSAnyType,
    NSIntType,
    NSPositiveIntType,
    NSFloatType,
    NSPositiveFloatType,
    NSDoubleType,
    NSPositiveDoubleType
};
```

DESCRIPTION These constants represent the numeric data types that a text NSCell can accept. See NSCell's **setEntryType:** method for more information.

NSCell—States

DECLARED IN AppKit/NSCell.h

SYNOPSIS

```
enum {
    NSStateMixed,
    NSStateOff,
    NSStateOn
};
```

DESCRIPTION These constants are suggested parameter values for the NSCell method **setState:**.

NSColorPanel—Modes

DECLARED IN AppKit/NSColorPanel.h

SYNOPSIS

```
enum {
    NSGrayModeColorPanel,
    NSRGBModeColorPanel,
    NSCMYKModeColorPanel,
    NSHSBModeColorPanel,
    NSCustomPaletteModeColorPanel,
    NSColorListModeColorPanel,
    NSWheelModeColorPanel
};
```

DESCRIPTION These constants represent the possible modes of an NSColorPanel.

NSColorPanel—Mode Masks

DECLARED IN AppKit/NSColorPanel.h

SYNOPSIS

```
enum {
    NSColorPanelGrayModeMask,
    NSColorPanelRGBModeMask,
    NSColorPanelCMYKModeMask,
    NSColorPanelHSBModeMask,
    NSColorPanelCustomPaletteModeMask,
    NSColorPanelColorListModeMask,
    NSColorPanelWheelModeMask,
    NSColorPanelAllModesMask
};
```

DESCRIPTION These constants provide masks for the NSColorPanel modes.

NSDragging—Operations

DECLARED IN AppKit/NSDragging.h

SYNOPSIS

```
enum {
    NSDragOperationNone,
    NSDragOperationCopy,
    NSDragOperationLink,
    NSDragOperationGeneric,
    NSDragOperationPrivate,
    NSDragOperationAll
};
```

DESCRIPTION These constants define the operations that result from a user's drag. For full descriptions of their meanings and uses, see the method descriptions for **draggingSourceOperationMaskForLocal:** (in the NSDraggingSource protocol), **draggingSourceOperationMask** (in the NSDraggingInfo protocol), or **draggingEntered:** (in the NSDraggingDestination protocol).

NSEvent—Function-Key Unicodes

DECLARED IN AppKit/NSEvent.h

SYNOPSIS

```
enum {
    NSUpArrowFunctionKey = 0xF700,
    NSDownArrowFunctionKey = 0xF701,
    NSLeftArrowFunctionKey = 0xF702,
    NSRightArrowFunctionKey = 0xF703,
    NSF1FunctionKey = 0xF704,
    NSF2FunctionKey = 0xF705,
    NSF3FunctionKey = 0xF706,
    NSF4FunctionKey = 0xF707,
    NSF5FunctionKey = 0xF708,
    NSF6FunctionKey = 0xF709,
    NSF7FunctionKey = 0xF70A,
    NSF8FunctionKey = 0xF70B,
    NSF9FunctionKey = 0xF70C,
    NSF10FunctionKey = 0xF70D,
```

NSF11FunctionKey = 0xF70E,
NSF12FunctionKey = 0xF70F,
NSF13FunctionKey = 0xF710,
NSF14FunctionKey = 0xF711,
NSF15FunctionKey = 0xF712,
NSF16FunctionKey = 0xF713,
NSF17FunctionKey = 0xF714,
NSF18FunctionKey = 0xF715,
NSF19FunctionKey = 0xF716,
NSF20FunctionKey = 0xF717,
NSF21FunctionKey = 0xF718,
NSF22FunctionKey = 0xF719,
NSF23FunctionKey = 0xF71A,
NSF24FunctionKey = 0xF71B,
NSF25FunctionKey = 0xF71C,
NSF26FunctionKey = 0xF71D,
NSF27FunctionKey = 0xF71E,
NSF28FunctionKey = 0xF71F,
NSF29FunctionKey = 0xF720,
NSF30FunctionKey = 0xF721,
NSF31FunctionKey = 0xF722,
NSF32FunctionKey = 0xF723,
NSF33FunctionKey = 0xF724,
NSF34FunctionKey = 0xF725,
NSF35FunctionKey = 0xF726,
NSInsertFunctionKey = 0xF727,
NSDeleteFunctionKey = 0xF728,
NSHomeFunctionKey = 0xF729,
NSBeginFunctionKey = 0xF72A,
NSEndFunctionKey = 0xF72B,
NSPageUpFunctionKey = 0xF72C,
NSPageDownFunctionKey = 0xF72D,
NSPrintScreenFunctionKey = 0xF72E,
NSScrollLockFunctionKey = 0xF72F,
NSPauseFunctionKey = 0xF730,
NSSysReqFunctionKey = 0xF731,
NSBreakFunctionKey = 0xF732,
NSResetFunctionKey = 0xF733,
NSStopFunctionKey = 0xF734,
NSMenuFunctionKey = 0xF735,
NSUserFunctionKey = 0xF736,
NSSystemFunctionKey = 0xF737,
NSPrintFunctionKey = 0xF738,

NSClearLineFunctionKey = 0xF739,
NSClearDisplayFunctionKey = 0xF73A,
NSInsertLineFunctionKey = 0xF73B,
NSDeleteLineFunctionKey = 0xF73C,
NSInsertCharFunctionKey = 0xF73D,
NSDeleteCharFunctionKey = 0xF73E,
NSPrevFunctionKey = 0xF73F,
NSNextFunctionKey = 0xF740,
NSSelectFunctionKey = 0xF741,
NSExecuteFunctionKey = 0xF742,
NSUndoFunctionKey = 0xF743,
NSRedoFunctionKey = 0xF744,
NSFindFunctionKey = 0xF745,
NSHelpFunctionKey = 0xF746,
NSModeSwitchFunctionKey = 0xF747
};

DESCRIPTION These Unicodes (0xF700-0xF8FF) are reserved for function keys on the keyboard. Combined in NSStrings, they may be used in the return value of the NSEvent methods **characters...**, and in parameters of the NSEvent method **keyEventWithType:...characters:...**.

NSEvent—Modifier Flags

DECLARED IN AppKit/NSEvent.h

SYNOPSIS

enum {
NSAlphaShiftKeyMask,
NSShiftKeyMask,
NSControlKeyMask,
NSAlternateKeyMask,
NSCommandKeyMask,
NSNumericPadKeyMask,
NSHelpKeyMask,
NSFunctionKeyMask
};

DESCRIPTION These are device-independent bits found in event modifier flags.

NSEvent—Types Defined by the Application Kit

DECLARED IN AppKit/NSEvent.h

SYNOPSIS

```
enum {
    NSWindowExposedEventType,
    NSApplicationActivatedEventType,
    NSApplicationDeactivatedEventType,
    NSWindowMovedEventType,
    NSScreenChangedEventType
};
```

DESCRIPTION These constants represent the types of events defined by the Application Kit.

Constant	Meaning
NSWindowExposedEventType	A nonretained NSWindow has been exposed
NSApplicationActivatedEventType	The application has been activated
NSApplicationDeactivatedEventType	The application has been deactivated
NSWindowMovedEventType	An NSWindow has moved
NSScreenChangedEventType	An NSWindow has changed screens

NSEvent—Types Defined by the System

DECLARED IN AppKit/NSEvent.h

SYNOPSIS

```
enum {
    NSPowerOffEventType
};
```

DESCRIPTION This constant means that the user is turning off the computer.

NSFont—Traits

DECLARED IN AppKit/NSFontManager.h

SYNOPSIS

```
enum {
    NSItalicFontMask,
    NSBoldFontMask,
    NSUnboldFontMask,
    NSNonStandardCharacterSetFontMask,
    NSNarrowFontMask,
    NSExpandedFontMask,
    NSCondensedFontMask,
    NSSmallCapsFontMask,
    NSPosterFontMask,
    NSCompressedFontMask,
    NSFixedPitchFontMask,
    NSUnitalicFontMask
};
```

DESCRIPTION These constants are used by the NSFontManager to identify font traits. Some traits are mutually exclusive, such as NSExpandedFontMask and NSCondensedFontMask.

NSFontPanel—Tags for Subviews

DECLARED IN AppKit/NSFontPanel.h

SYNOPSIS

```
enum {
    NSFPPreviewButton,
    NSFPRevertButton,
    NSFPSetButton,
    NSFPPreviewField,
    NSFPSizeField,
    NSFPSizeTitle,
    NSFPCurrentField
};
```

DESCRIPTION These tags identify the NSViews within an NSFontPanel.

NSGlyph—Attributes

DECLARED IN AppKit/NSLayoutManager.h

SYNOPSIS

```
enum _NSGlyphAttribute {
    NSGlyphAttributeSoft,
    NSGlyphAttributeElastic,
    NSGlyphAttributeInscribe
};
```

DESCRIPTION These glyph attributes are used only inside the glyph generation machinery, but must be shared between components.

NSGlyph—Reserved Glyph Codes

DECLARED IN AppKit/NSFont.h

SYNOPSIS

```
enum {
    NSControlGlyph = 0x00FFFFFF,
    NSNullGlyph = 0x0
};
```

DESCRIPTION These two values are reserved for the two named NSGlyphs.

NSImageRep—Display Device Matching

DECLARED IN AppKit/NSImageRep.h

SYNOPSIS

```
enum {
    NSImageRepMatchesDevice
};
```

DESCRIPTION This constant is used by NSImageRep to indicate that the value of certain attributes, such as the number of colors, or bits-per-sample, will change to match the display device. See the NSImageRep class specification for more information.

NSPageLayoutPanel—Tags for Controls

DECLARED IN AppKit/NSPageLayout.h

SYNOPSIS

```
enum {
    NSPLImageButton,
    NSPLTitleField,
    NSPLPaperNameButton,
    NSPLUnitsButton,
    NSPLWidthForm,
    NSPLHeightForm,
    NSPLOrientationMatrix,
    NSPLCancelButton,
    NSPLOKButton
};
```

DESCRIPTION These constants represent the tag values of the controls displayed by an NSPageLayoutPanel.

NSPanel—Alert Panel Return Values

DECLARED IN AppKit/NSPanel.h

SYNOPSIS

```
enum {
    NSAlertDefaultReturn,
    NSAlertAlternateReturn,
    NSAlertOtherReturn,
    NSAlertErrorReturn
};
```

DESCRIPTION These constants define values returned by the **NSRunAlertPanel** function and by the NSApplication method **runModalSession:** when the modal session is run with an NSPanel provided by the **NSGetAlertPanel** function.

NSPanel—Modal Panel Return Values

DECLARED IN AppKit/NSPanel.h

SYNOPSIS

```
enum {
    NSOKButton,
    NSCancelButton
};
```

DESCRIPTION These are the possible return values for such methods as the **runModal...** methods of NSOpenPanel, which tells which button (OK or Cancel) the user has clicked on an open panel. For other uses of these return values, see the class descriptions for NSPageLayout, NSPrintPanel and NSSavePanel.

NSPrintPanel—Tags for Subviews

DECLARED IN AppKit/NSPrintPanel.h

SYNOPSIS

```
enum {
    NSPPSaveButton,
    NSPPPreviewButton,
    NSFaxButton,
    NSPPTitleField,
    NSPPImageButton,
    NSPPNameTitle,
    NSPPNameField,
    NSPPNoteTitle,
    NSPPNoteField,
    NSPPStatusTitle,
    NSPPStatusField,
    NSPPCopiesField,
    NSPPPageChoiceMatrix,
    NSPPPageRangeFrom,
    NSPPPageRangeTo,
    NSPPScaleField,
    NSPPOptionsButton,
    NSPPPaperFeedButton,
    NSPPLayoutButton
};
```

DESCRIPTION These constants define tags for identifying the NSViews in a print panel in environments other than Microsoft Windows. Windows has its own way of handling print panels.

NSRunLoop—Ordering Modes for NSApplication

DECLARED IN AppKit/NSApplication.h

SYNOPSIS

```
enum {
    NSUpdateWindowsRunLoopOrdering
};
```

DESCRIPTION This constants is used with NSRunLoop's **performSelector:target:argument:order:modes:** method.

NSRunLoop—Ordering Mode for NSDPSServerContext

DECLARED IN AppKit/NSDPSServerContext.h

SYNOPSIS

```
enum {
    DPSFlushContextRunLoopOrdering
};
```

DESCRIPTION This constants is used with NSRunLoop's method **performSelector:target:argument:order:modes:**.

NSRunLoop—Ordering Modes for NSWindow

DECLARED IN AppKit/NSWindow.h

SYNOPSIS

```
enum {
    NSDisplayWindowRunLoopOrdering,
    NSResetCursorRectsRunLoopOrdering
};
```

DESCRIPTION These constants are passed to NSRunLoop’s method **performSelector:target:argument:order:modes:**.

NSSavePanel—Tags for Subviews

DECLARED IN AppKit/NSSavePanel.h

SYNOPSIS

```
enum {
    NSFileHandlingPanelImageButton,
    NSFileHandlingPanelTitleField,
    NSFileHandlingPanelBrowser,
    NSFileHandlingPanelCancelButton,
    NSFileHandlingPanelOKButton,
    NSFileHandlingPanelForm,
    NSFileHandlingPanelHomeButton,
    NSFileHandlingPanelDiskButton,
    NSFileHandlingPanelDiskEjectButton
};
```

DESCRIPTION These constants define tags for identifying NSViews in an NSSavePanel.

NSTextAttachment—Attachment Character

DECLARED IN AppKit/NSTextAttachment.h

SYNOPSIS

```
enum {
    NSAttachmentCharacter = 0xfffc
};
```

DECLARED IN This Unicode indicates the presence of an attachment in an NSAttributedString. For more information, see the Class Cluster Description of NSAttributedStringAdditions.

NSText—Important Unicodes

DECLARED IN AppKit/NSText.h

SYNOPSIS

```
enum {
    NSParagraphSeparatorCharacter = 0x2029,
    NSLineSeparatorCharacter = 0x2028,
    NSTabCharacter = 0x0009,
    NSFormFeedCharacter = 0x000c,
    NSNewlineCharacter = 0x000a,
    NSCarriageReturnCharacter = 0x000d,
    NSEnterCharacter = 0x0003,
    NSBackspaceCharacter = 0x0008,
    NSBackTabCharacter = 0x0019,
    NSDeleteCharacter = 0x007f,
};
```

DESCRIPTION These constants specify several commonly-used Unicode characters.

NSText—Movement Codes

DECLARED IN AppKit/NSText.h

SYNOPSIS

```
enum {
    NSIllegalTextMovement,
    NSReturnTextMovement,
    NSTabTextMovement,
    NSBacktabTextMovement,
    NSLeftTextMovement,
    NSRightTextMovement,
    NSUpTextMovement,
    NSDownTextMovement
};
```

DESCRIPTION These constants are the codes for movement between fields. They are the possible int values for the NSTextMovement key of NSTextDidEndEditingNotification. For more information, see the "Notifications" section of the NSText class specification.

NSTextStorage—Editing

DECLARED IN AppKit/NSTextStorage.h

SYNOPSIS

```
enum {
    NSTextStorageEditedAttributes,
    NSTextStorageEditedCharacters
};
```

DESCRIPTION These values, which may be combined by a bitwise OR, help describe the changes that an editing session has made to an NSTextStorage object. They are the return values of the NSTextStorage method **editedMask**, and the parameter values for the second slot of the NSLayoutManager method **textStorage:edited:...**.

NSView—Resizing

DECLARED IN AppKit/NSView.h

SYNOPSIS

```
enum {
    NSViewNotSizable,
    NSViewMinXMargin,
    NSViewWidthSizable,
    NSViewMaxXMargin,
    NSViewMinYMargin,
    NSViewHeightSizable,
    NSViewMaxYMargin
};
```

DESCRIPTION Used to describe which parts of an NSView (or its margins) are resized when the NSView's superNSView is resized. See the NSView class specification for details.

NSWindow—Border Masks

DECLARED IN AppKit/NSWindow.h

SYNOPSIS

```
enum {
    NSBorderlessWindowMask,
    NSTitledWindowMask,
    NSClosableWindowMask,
    NSMiniaturizableWindowMask,
    NSResizableWindowMask
};
```

DESCRIPTION These determine the presence of a title and various buttons in an NSWindow's border.

NSWindow—Window Levels

DECLARED IN AppKit/NSWindow.h

SYNOPSIS

```
enum {
    NSNormalWindowLevel,
    NSFloatingWindowLevel,
    NSDockWindowLevel,
    NSSubmenuWindowLevel,
    NSTornOffMenuWindowLevel,
    NSMainMenuWindowLevel,
    NSModalPanelWindowLevel,
    NSPopUpMenuWindowLevel
};
```

DESCRIPTION These constants name the Application Kit's window levels. The stacking of levels takes precedence over the stacking of windows within each level. That is, even the bottom window in a level will obscure even the top window of the next level down.

The constant NSTornOffMenuWindowLevel is preferable to its synonym, NSSubmenuWindowLevel.

Global Variables

Application Kit—Exceptions

DECLARED IN AppKit/NSErrors.h

SYNOPSIS

NSString ***NSTextLineTooLongException**;
NSString ***NSTextNoSelectionException**;
NSString ***NSWordTablesWriteException**;
NSString ***NSWordTablesReadException**;
NSString ***NSTextReadException**;
NSString ***NSTextWriteException**;
NSString ***NSPasteboardCommunicationException**;
NSString ***NSPrintingCommunicationException**;
NSString ***NSAbortModalException**;
NSString ***NSAbortPrintingException**;
NSString ***NSIllegalSelectorException**;
NSString ***NSAppKitVirtualMemoryException**;
NSString ***NSBadRTFDirectiveException**;
NSString ***NSBadRTFFontTableException**;
NSString ***NSBadRTFStyleSheetException**;
NSString ***NSTypedStreamVersionException**;
NSString ***NSTIFFException**;
NSString ***NSPrintPackageException**;
NSString ***NSBadRTFColorTableException**;
NSString ***NSDraggingException**;

NSString ***NSColorListIOException**;
NSString ***NSColorListNotEditableException**;
NSString ***NSBadBitmapParametersException**;
NSString ***NSWindowServerCommunicationException**;
NSString ***NSFontUnavailableException**;
NSString ***NSPPDIncludeNotFoundException**;
NSString ***NSPPDParseException**;
NSString ***NSPPDIncludeStackOverflowException**;
NSString ***NSPPDIncludeStackUnderflowException**;
NSString ***NSRTFPropertyStackOverflowException**;
NSString ***NSAppKitIgnoredException**;
NSString ***NSBadComparisonException**;
NSString ***NSImageCacheException**;
NSString ***NSNibLoadingException**;
NSString ***NSBrowserIllegalDelegateException**;

DESCRIPTION These constants name the exceptions that the Application Kit can raise.

Display Device—Descriptions

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS
NSString ***NSDeviceResolution**;
NSString ***NSDeviceColorSpaceName**;
NSString ***NSDeviceBitsPerSample**;
NSString ***NSDeviceIsScreen**;
NSString ***NSDeviceIsPrinter**;
NSString ***NSDeviceSize**;

DESCRIPTION These are the keys for device description dictionaries, such as those returned by the **deviceDictionary** methods of NSPrinter, NSScreen and NSWindow.

NSDeviceResolution is an NSValue containing an NSSize in dots per inch. NSColorSpaceName is an NSString describing the color space of the device. NSDeviceBitsPerSample is an NSValue containing an int. NSDeviceIsScreen and NSDeviceIsPrinter are boolean values that tell whether the device is a screen or a printer. NSDeviceSize is an NSValue containing an NSSize that represents the device's size in points.

NSApplication—Notifications

DECLARED IN AppKit/NSApplication.h

SYNOPSIS

NSString ***NSApplicationDidBecomeActiveNotification**;
NSString ***NSApplicationDidFinishLaunchingNotification**;
NSString ***NSApplicationDidHideNotification**;
NSString ***NSApplicationDidResignActiveNotification**;
NSString ***NSApplicationDidUnhideNotification**;
NSString ***NSApplicationDidUpdateNotification**;
NSString ***NSApplicationWillBecomeActiveNotification**;
NSString ***NSApplicationWillFinishLaunchingNotification**;
NSString ***NSApplicationWillHideNotification**;
NSString ***NSApplicationWillResignActiveNotification**;
NSString ***NSApplicationWillUnhideNotification**;
NSString ***NSApplicationWillUpdateNotification**;
NSString ***NSApplicationWillTerminateNotification**;

DECLARED IN These are the notifications used with the methods of the NSApplicationNotifications category of NSObject.

NSApplication—Shared Application Object

DECLARED IN AppKit/NSApplication.h

SYNOPSIS id **NSApp**;

DESCRIPTION This variable designates the shared application object, created by NSApplication's **sharedApplication** method.

NSAttributedString—Attributes

DECLARED IN AppKit/NSAttributedString.h

SYNOPSIS

NSString ***NSFontAttributeName**;
NSString ***NSParagraphStyleAttributeName**;
NSString ***NSForegroundColorAttributeName**;
NSString ***NSUnderlineStyleAttributeName**;
NSString ***NSSuperscriptAttributeName**;
NSString ***NSBackgroundColorAttributeName**;
NSString ***NSAttachmentAttributeName**;
NSString ***NSLigatureAttributeName**;
NSString ***NSBaselineOffsetAttributeName**;
NSString ***NSKernAttributeName**;

DESCRIPTION These strings define the supported attributes of NSAttributedStrings. For more information, see the “Accessing Attributes” section in the NSAttributedString class cluster specification.

NSComboBox—Notifications

DECLARED IN AppKit/NSComboBox.h

SYNOPSIS

NSString ***NSComboBoxWillPopUpNotification**;
NSString ***NSComboBoxWillDismissNotification**;
NSString ***NSComboBoxSelectionDidChangeNotification**;
NSString ***NSComboBoxSelectionIsChangingNotification**;

DESCRIPTION These notifications are sent by NSComboBoxes.

NSColor—Color Space Names

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS

NSString ***NSCalibratedWhiteColorSpace**;
NSString ***NSCalibratedBlackColorSpace**;
NSString ***NSCalibratedRGBColorSpace**;
NSString ***NSDeviceWhiteColorSpace**;
NSString ***NSDeviceBlackColorSpace**;
NSString ***NSDeviceRGBColorSpace**;
NSString ***NSDeviceCMYKColorSpace**;
NSString ***NSNamedColorSpace**;
NSString ***NSCustomColorSpace**;

DESCRIPTION These are the predefined names for color spaces. In the two ...WhiteColorSpaces, white corresponds to a value of 1.0. In the two ...BlackColorSpaces, black corresponds to a value of 1.0. NSNamedColorSpace is used for “catalog” colors—that is, colors specified by names rather than coordinates. NSCustomColorSpace indicates a custom color space, which can be useful in working with images; unlike the other color spaces, NSCustomColorSpace is not used with NSColors.

NSColor—Grayscale Values

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS

const float **NSWhite**;
const float **NSLightGray**;
const float **NSDarkGray**;
const float **NSBlack**;

DESCRIPTION These are the standard gray values for the 2-bit deep grayscale color space.

NSColor—Notifications

DECLARED IN AppKit/NSColor.h

SYNOPSIS NSString ***NSSystemColorsDidChangeNotification**;

DESCRIPTION This notification is sent when the system colors have been changed (such as through a system control panel interface). For more on system colors, see the “System Colors” section of the NSColor class specification.

NSColorList—Notifications

DECLARED IN AppKit/NSColorList.h

SYNOPSIS NSString ***NSColorListDidChangeNotification**;

DESCRIPTION When an NSColorList changes, it posts this notification.

NSColorPanel—Notifications

DECLARED IN AppKit/NSColorPanel.h

SYNOPSIS NSString ***NSColorPanelColorDidChangeNotification**;

DESCRIPTION When an NSColorPanel changes, it posts this notification.

NSControl—Notifications

DECLARED IN AppKit/NSControl.h

SYNOPSIS NSString ***NSControlTextDidBeginEditingNotification**;
NSString ***NSControlTextDidEndEditingNotification**;
NSString ***NSControlTextDidChangeNotification**;

DESCRIPTION NSControls containing editable text can send these notifications. For more information, see the "Notifications" section of the NSControl class specification.

NSDataLink—Filename Extension

DECLARED IN AppKit/NSDataLink.h

SYNOPSIS NSString ***NSDataLinkFilenameExtension**;

DESCRIPTION NSDataLinkFilenameExtension is the filename extension used for links saved to files using NSDataLink's **saveLinkIn:** or **writeToFile:** methods.

NSFont—Keys to the AFM Dictionary

DECLARED IN AppKit/NSFont.h

SYNOPSIS

NSString ***NSAFMFamilyName**;
NSString ***NSAFMFontName**;
NSString ***NSAFMFormatVersion**;
NSString ***NSAFMFullName**;
NSString ***NSAFMNotice**;
NSString ***NSAFMVersion**;
NSString ***NSAFMWeight**;
NSString ***NSAFMEncodingScheme**;
NSString ***NSAFMCharacterSet**;
NSString ***NSAFMCapHeight**;
NSString ***NSAFMXHeight**;
NSString ***NSAFMAscender**;
NSString ***NSAFMDescender**;
NSString ***NSAFMUnderlinePosition**;
NSString ***NSAFMUnderlineThickness**;
NSString ***NSAFMItalicAngle**;
NSString ***NSAFMMappingScheme**;

DESCRIPTION These are the keys to the font information dictionary returned by NSFont's **afmDictionary** method. To convert values like NSAFMCapHeight to floats, use NSString's **floatValue** method.

For other font information, use NSFont's **afmFileContents** method.

NSFont—PostScript Transformation Matrix

DECLARED IN AppKit/NSFont.h

SYNOPSIS

const float ***NSFontIdentityMatrix**;

DESCRIPTION NSFontIdentityMatrix is a PostScript transformation matrix useful as a parameter to the NSFont method **fontWithName:matrix:**.

NSHelpManager—Notifications

DECLARED IN AppKit/NSHelpManager.h

SYNOPSIS NSString ***NSContextHelpModeDidActivateNotification**;
NSString ***NSContextHelpModeDidDeactivateNotification**;

DESCRIPTION These are notifications for the activation and deactivation of the context help mode.

NSImageRep—Notifications

DECLARED IN AppKit/NSImageRep.h

SYNOPSIS NSString ***NSImageRepRegistryDidChangeNotification**;

DESCRIPTION This notification is sent when the NSImageRep class registry changes.

NSInterfaceStyleDefault

DECLARED IN AppKit/NSInterfaceStyle.h

SYNOPSIS NSString ***NSInterfaceStyleDefault**;

DESCRIPTION NSInterfaceStyleDefault can be used to override the platform's default interface style. For more information, see the function **NSInterfaceStyleForKey**.

NSPasteboard—Names

DECLARED IN AppKit/NSPasteboard.h

SYNOPSIS
NSString ***NSGeneralPboard**;
NSString ***NSFontPboard**;
NSString ***NSRulerPboard**;
NSString ***NSFindPboard**;
NSString ***NSDragPboard**;

DESCRIPTION Some standard pasteboard names. See the NSPasteboard class specification for more information.

NSPasteboard—Type for Data Links

DECLARED IN AppKit/NSDataLink.h

SYNOPSIS NSString ***NSDataLinkPboardType**;

DESCRIPTION A pasteboard type for copying a data link to the pasteboard. See the NSDataLink class specification for more information.

NSPasteboard—Type for Selection Descriptions

DECLARED IN AppKit/NSSelection.h

SYNOPSIS NSString ***NSSelectionPboardType**;

DESCRIPTION A pasteboard type for copying selection descriptions to the pasteboard. See the NSSelection class specification for more information.

NSPasteboard—Types for Standard Data

DECLARED IN AppKit/NSPasteboard.h

SYNOPSIS

NSString ***NSStringPboardType**;
NSString ***NSFilenamesPboardType**;
NSString ***NSPostScriptPboardType**;
NSString ***NSTIFFPboardType**;
NSString ***NSRTFPboardType**;
NSString ***NSTabularTextPboardType**;
NSString ***NSFontPboardType**;
NSString ***NSRulerPboardType**;
NSString ***NSFileContentsPboardType**;
NSString ***NSColorPboardType**;
NSString ***NSRTFDPboardType**;

DESCRIPTION Some standard pasteboard data types. See the NSPasteboard class specification for more information.

NSPrintInfo—Dictionary Keys

DECLARED IN AppKit/NSPrintInfo.h

SYNOPSIS

NSString ***NSPrintPaperName**;
NSString ***NSPrintPaperSize**;
NSString ***NSPrintFormName**;
NSString ***NSPrintMustCollate**;
NSString ***NSPrintOrientation**;
NSString ***NSPrintLeftMargin**;
NSString ***NSPrintRightMargin**;
NSString ***NSPrintTopMargin**;
NSString ***NSPrintBottomMargin**;
NSString ***NSPrintHorizontallyCentered**;
NSString ***NSPrintVerticallyCentered**;
NSString ***NSPrintHorizontalPagination**;
NSString ***NSPrintVerticalPagination**;
NSString ***NSPrintScalingFactor**;
NSString ***NSPrintAllPages**;
NSString ***NSPrintReversePageOrder**;
NSString ***NSPrintFirstPage**;
NSString ***NSPrintLastPage**;
NSString ***NSPrintCopies**;
NSString ***NSPrintPagesPerSheet**;
NSString ***NSPrintJobFeatures**;
NSString ***NSPrintPaperFeed**;
NSString ***NSPrintManualFeed**;
NSString ***NSPrintPrinter**;
NSString ***NSPrintJobDisposition**;

NSString ***NSPrintSavePath**;
NSString ***NSPrintFaxReceiverNames**;
NSString ***NSPrintFaxReceiverNumbers**;
NSString ***NSPrintFaxSendTime**;
NSString ***NSPrintFaxUseCoverSheet**;
NSString ***NSPrintFaxCoverSheetName**;
NSString ***NSPrintFaxReturnReceipt**;
NSString ***NSPrintFaxHighResolution**;
NSString ***NSPrintFaxTrimPageEnds**;
NSString ***NSPrintFaxModem**;
NSString ***NSPrintSpoolJob**;
NSString ***NSPrintFaxJob**;
NSString ***NSPrintPreviewJob**;
NSString ***NSPrintSaveJob**;
NSString ***NSPrintCancelJob**;

DESCRIPTION These are the keys to the NSPrintInfo NSDictionary. For a table explaining them, see the NSPrintInfo method **initWithDictionary:**.

NSPopUpButton—Notification

DECLARED IN AppKit/NSPopUpButton.h

SYNOPSIS NSString ***NSPopUpButtonWillPopUpNotification**;

DESCRIPTION NSPopUpButton sends this notification when an instance of it is about to pop up.

NSPrintOperation—Exception

DECLARED IN AppKit/NSPrintOperation.h

SYNOPSIS NSString ***NSPrintOperationExistsException**;

DESCRIPTION This exception is raised when there is already a print operation in process. The methods that raise it are the **EPSOperation...** and **printOperation...** methods in NSPrintOperation:

NSRunLoop—Modes

DECLARED IN AppKit/NSApplication.h

SYNOPSIS NSString ***NSModalPanelRunLoopMode**;
NSString ***NSEventTrackingRunLoopMode**;

DESCRIPTION These are modes passed to NSRunLoop

NSSplitView—Notifications

DECLARED IN AppKit/NSSplitView.h

SYNOPSIS NSString ***NSSplitViewDidResizeSubviewsNotification**;
NSString ***NSSplitViewWillResizeSubviewsNotification**;

DESCRIPTION These are the notifications that an NSSplitView can send.

NSTableView—Notifications

DECLARED IN AppKit/NSTableView.h

SYNOPSIS NSString ***NSTableViewSelectionDidChangeNotification**;
NSString ***NSTableViewColumnDidMoveNotification**;
NSString ***NSTableViewColumnDidResizeNotification**;
NSString ***NSTableViewSelectionIsChangingNotification**;

DESCRIPTION These are the notifications that an NSTableView can send.

NSText—Notifications

DECLARED IN AppKit/NSText.h

SYNOPSIS NSString ***NSTextDidBeginEditingNotification**;
NSString ***NSTextDidEndEditingNotification**;
NSString ***NSTextDidChangeNotification**;

DESCRIPTION These notifications can be sent by an NSText object. For explanations, see the “Notifications” section of the NSText class specification.

NSTextStorage—Notifications

DECLARED IN AppKit/NSTextStorage.h

SYNOPSIS NSString ***NSTextStorageWillProcessEditingNotification**;
NSString ***NSTextStorageDidProcessEditingNotification**;

DESCRIPTION These notifications can be sent by an NSTextStorage object. For explanations, see the “Notifications” section of the NSTextStorage class specification.

NSTextView—Notifications

DECLARED IN AppKit/NSTextView.h

SYNOPSIS NSString ***NSTextViewWillChangeNotifyingTextViewNotification**;
NSString ***NSTextViewDidChangeSelectionNotification**;

DESCRIPTION These notifications can be sent by an NSTextView object. For explanations, see the “Notifications” section of the NSTextView class specification.

The notifications that NSTextView most often sends are the ones that it inherits from NSText.

NSView—Notifications

DECLARED IN AppKit/NSView.h

SYNOPSIS
NSString ***NSViewFrameDidChangeNotification**;
NSString ***NSViewFocusDidChangeNotification**;
NSString ***NSViewBoundsDidChangeNotification**;

DESCRIPTION These notifications are sent by NSViews.

The last notification, NSViewBoundsDidChangeNotification, is sent when the view bounds change but the frame does not. That is, it is sent whenever the view's bounds are translated, scaled or rotated, but not when the bounds change in response to, say, a **setFrameSize:** message.

NSWindow—Notifications

DECLARED IN AppKit/NSWindow.h

SYNOPSIS
NSString ***NSWindowDidBecomeKeyNotification**;
NSString ***NSWindowDidBecomeMainNotification**;
NSString ***NSWindowDidChangeScreenNotification**;
NSString ***NSWindowDidDeminiaturizeNotification**;
NSString ***NSWindowDidExposeNotification**;
NSString ***NSWindowDidMiniaturizeNotification**;
NSString ***NSWindowDidMoveNotification**;
NSString ***NSWindowDidResignKeyNotification**;
NSString ***NSWindowDidResignMainNotification**;
NSString ***NSWindowDidResizeNotification**;
NSString ***NSWindowDidUpdateNotification**;
NSString ***NSWindowWillCloseNotification**;
NSString ***NSWindowWillMiniaturizeNotification**;
NSString ***NSWindowWillMoveNotification**;

DESCRIPTION These are the notifications that can be sent by an NSWindow object. For explanations, see the “Notifications” section of the NSWindow class specification.

NSWindow—Sizes

DECLARED IN AppKit/NSWindow.h

SYNOPSIS

NSSize **NSIconSize**;
NSSize **NSTokenSize**;

DESCRIPTION On some platforms, a token is a bezeled tile used to represent a docked application or a miniaturized document, and an icon is the image drawn inside a token.

On platforms that support tokens and icons, these size constants can be used for drawing inside them. It is more portable, however, to change an icon by using the NSApplication method **setApplicationIconImage:** or the NSWindow method **setMiniwindowImage:**.

NSWorkspace—File Operation Constants

DECLARED IN AppKit/NSWorkspace.h

SYNOPSIS

NSString ***NSWorkspaceMoveOperation**;
NSString ***NSWorkspaceCopyOperation**;
NSString ***NSWorkspaceLinkOperation**;
NSString ***NSWorkspaceCompressOperation**;
NSString ***NSWorkspaceDecompressOperation**;
NSString ***NSWorkspaceEncryptOperation**;
NSString ***NSWorkspaceDecryptOperation**;
NSString ***NSWorkspaceDestroyOperation**;
NSString ***NSWorkspaceRecycleOperation**;
NSString ***NSWorkspaceDuplicateOperation**;

DESCRIPTION These constants define possible values for the *operation* slot in NSWorkspace's **performFileOperation:**... method.

NSWorkspace—File Types

DECLARED IN AppKit/NSWorkspace.h

SYNOPSIS
NSString ***NSPlainFileType**;
NSString ***NSDirectoryFileType**;
NSString ***NSApplicationFileType**;
NSString ***NSFilesystemFileType**;
NSString ***NSShellCommandFileType**;

DESCRIPTION These values are used in the final parameter slot of the NSWorkspace method **getInfoForFile: application:type:**.

NSWorkspace—Notifications

DECLARED IN AppKit/NSWorkspace.h

SYNOPSIS
NSString ***NSWorkspaceDidLaunchApplicationNotification**;
NSString ***NSWorkspaceDidMountNotification**;
NSString ***NSWorkspaceDidPerformFileOperationNotification**;
NSString ***NSWorkspaceDidTerminateApplicationNotification**;
NSString ***NSWorkspaceDidUnmountNotification**;
NSString ***NSWorkspaceWillLaunchApplicationNotification**;
NSString ***NSWorkspaceWillPowerOffNotification**;
NSString ***NSWorkspaceWillUnmountNotification**;

DESCRIPTION These notifications come through the special notification center. For more information, see the "Notifications" section of the NSWorkspace class specification.

Defined Types

NSBorderType

DECLARED IN AppKit/NSView.h

SYNOPSIS

```
typedef enum _NSBorderType {
    NSNoBorder,
    NSLineBorder,
    NSBezelBorder,
    NSGrooveBorder
} NSBorderType;
```

DESCRIPTION This type represents the kinds of border that can be drawn around certain NSView subclasses.

NSButtonType

DECLARED IN AppKit/NSButtonCell.h

SYNOPSIS

```
typedef enum _NSButtonType {
    NSMomentaryPushButton,
    NSPushOnPushOffButton,
    NSToggleButton,
    NSSwitchButton,
    NSRadioButton,
    NSMomentaryChangeButton,
    NSOnOffButton,
    NSMomentaryLight,
} NSButtonType;
```

DESCRIPTION This type represents the way NSButtons and NSButtonCells behave when pressed, and the way they display their state. See NSButton's and NSButtonCell's **setButtonType:** methods for more information.

NSCellAttribute

DECLARED IN AppKit/NSCell.h

SYNOPSIS

```
typedef enum _NSCellAttribute {
    NSCellDisabled,
    NSCellState,
    NSPushInCell,
    NSCellEditable,
    NSChangeGrayCell,
    NSCellHighlighted,
    NSCellLightsByContents,
    NSCellLightsByGray,
    NSChangeBackgroundCell,
    NSCellLightsByBackground,
    NSCellIsBordered,
    NSCellHasOverlappingImage,
    NSCellHasImageHorizontal,
    NSCellHasImageOnLeftOrBottom,
    NSCellChangesContents,
    NSCellIsInsetButton,
    NSCellAllowsMixedState
} NSCellAttribute;
```

DESCRIPTION This is the type of the first argument to the NSCell methods **setCellAttribute:to:** and **cellAttribute:** methods. Some of the values apply not to NSCell but to one of its subclasses.

Often it's preferable to change cell attributes using more specialized methods like **setState:** or **setEditable:**.

NSCellImagePosition

DECLARED IN AppKit/NSCell.h

SYNOPSIS

typedef enum _NSCellImagePosition {
    NSNoImage,
    NSImageOnly,
    NSImageLeft,
    NSImageRight,
    NSImageBelow,
    NSImageAbove,
    NSImageOverlaps,
} **NSCellImagePosition**;

DESCRIPTION These constants represent the position of an NSButtonCell's NSImage relative to its title. See NSButton's and NSButtonCell's **setImagePosition:** and **imagePosition** methods for more information.

NSCellType

DECLARED IN AppKit/NSCell.h

SYNOPSIS

typedef enum _NSCellType {
    NSNullCellType,
    NSTextCellType,
    NSImageCellType,
} **NSCellType**;

DESCRIPTION The values of this type determine what kind of data an NSCell displays. NSCells of type NSTextCellType and NSImageCellType display text and images, respectively. NSCells of type NSNullCellType display nothing. See the NSCell methods **type** and **setType:** for more information.

NSDataLinkDisposition

DECLARED IN AppKit/NSDataLink.h

SYNOPSIS

```
typedef enum _NSDataLinkDisposition {
    NSLinkInDestination,
    NSLinkInSource,
    NSLinkBroken
} NSDataLinkDisposition;
```

DESCRIPTION Returned by NSDataLink's **disposition** method to identify a link as a destination link, a source link, or a broken link. See the NSDataLink class specification for more information on the dispositions of links.

NSDataLinkNumber

DECLARED IN AppKit/NSDataLink.h

SYNOPSIS

```
typedef int NSDataLinkNumber;
```

DESCRIPTION The type returned by NSDataLink's **linkNumber** method as a persistent identifier of a destination link.

NSDataLinkUpdateMode

DECLARED IN AppKit/NSDataLink.h

SYNOPSIS

```
typedef enum _NSDataLinkUpdateMode {
    NSUpdateContinuously,
    NSUpdateWhenSourceSaved,
    NSUpdateManually,
    NSUpdateNever
} NSDataLinkUpdateMode;
```

DESCRIPTION Used by NSDataLink's **setUpdateMode:** and **updateMode** methods to identify when a link's data is to be updated.

NSEventType

DECLARED IN AppKit/NSEvent.h

SYNOPSIS

```
typedef enum _NSEventType {
        NSLeftMouseDown,
        NSLeftMouseUp,
        NSRightMouseDown,
        NSRightMouseUp,
        NSMouseMoved,
        NSLeftMouseDragged,
        NSRightMouseDragged,
        NSMouseEntered,
        NSMouseExited,
        NSKeyDown,
        NSKeyUp,
        NSFlagsChanged,
        NSAppKitDefined,
        NSSystemDefined,
        NSApplicationDefined
        NSPeriodic,
        NSCursorUpdate
} NSEventType;
```

DESCRIPTION This type represents various kinds of events. It is the return type of NSEvent's type method, and the type of the first argument to NSEvent's **...EventWithType:** methods.

NSFontAction

DECLARED IN AppKit/NSFontManager.h

SYNOPSIS

typedef enum _NSFontAction {
    NSNoFontChangeAction,
    NSViaPanelFontAction,
    NSAddTraitFontAction,
    NSSizeUpFontAction,
    NSSizeDownFontAction,
    NSHeavierFontAction,
    NSLighterFontAction,
    NSRemoveTraitFontAction
} **NSFontAction**;

DESCRIPTION Values of this type tag the actions of font menu cells. When a font menu cell sends a message to NSFontManager, NSFontManager checks the cell for one of these tags.

This type is in the API for explanatory purposes only. You will never use it directly.

NSFontTraitMask

DECLARED IN AppKit/NSFontManager.h

SYNOPSIS typedef unsigned int **NSFontTraitMask;**

DESCRIPTION An NSFontTraitMask characterizes one or more of a font's traits. It's used as an argument type for NSAttributedString's **applyFontTraits:range:** method, NSCStringText's **setSelFontStyle:** method, and several of the methods in the NSFontManager class.

NSGlyph

DECLARED IN AppKit/NSFont.h

SYNOPSIS typedef unsigned int **NSGlyph**;

DESCRIPTION This type is used to specify PostScript glyphs in such NSFont methods as **glyphWithName:**.

NSGlyphInscription

DECLARED IN AppKit/NSLayoutManager.h

SYNOPSIS

```
typedef enum {
    NSGlyphInscribeBase,
    NSGlyphInscribeBelow,
    NSGlyphInscribeAbove,
    NSGlyphInscribeOverstrike,
    NSGlyphInscribeOverBelow
} NSGlyphInscription;
```

DESCRIPTION The inscribe attribute of an glyph determines how it is laid out relative to the previous glyph.

NSGlyphRelation

DECLARED IN AppKit/NSFont.h

SYNOPSIS

```
typedef enum _NSGlyphRelation {
    NSGlyphBelow,
    NSGlyphAbove,
} NSGlyphRelation;
```

DESCRIPTION This type specifies the position of a glyph in relation to the base glyph. Parameters of this type are used in the second slot of the NSFont method **positionOfGlyph:withRelation: toBaseGlyph:...**.

NSGradientType

DECLARED IN AppKit/NSButtonCell.h

SYNOPSIS

typedef enum _NSGradientType {
    NSGradientNone,
    NSGradientConcaveWeak,
    NSGradientConcaveStrong,
    NSGradientConvexWeak,
    NSGradientConvexStrong
} **NSGradientType**;

DESCRIPTION This type represents the darkness gradient of an NSButtonCell. A concave gradient is darkest in the top left corner, a convex gradient is darkest in the bottom right corner. A weak gradient has only weak darkness contrast between opposite corners; a strong gradient has strong contrast. See the NSButtonCell methods **gradient** and **setGradient:** for more information.

NSImageAlignment

DECLARED IN AppKit/NSImageCell.h

SYNOPSIS

typedef enum {
    NSImageAlignCenter,
    NSImageAlignTop,
    NSImageAlignTopLeft,
    NSImageAlignTopRight,
    NSImageAlignLeft,
    NSImageAlignBottom,
    NSImageAlignBottomLeft,
    NSImageAlignBottomRight,
    NSImageAlignRight
} **NSImageAlignment**;

DESCRIPTION This type defines the ways of aligning an NSImage within an NSImageCell. It is the return type for NSImageCell's and NSImageView's **imageAlignment** methods, and an argument type for their **setImageAlignment:** methods.

NSImageFrameStyle

DECLARED IN AppKit/NSImageCell.h

SYNOPSIS

```
typedef enum {
    NSImageFrameNone,
    NSImageFramePhoto,
    NSImageFrameGrayBezel,
    NSImageFrameGroove,
    NSImageFrameButton
} NSImageFrameStyle;
```

DESCRIPTION This type defines the kinds of frames that can appear around an NSImageCell. It is the return type for NSImageCell's and NSImageView's **imageFrameStyle** methods, and an argument type for their **setImageFrameStyle:** methods.

NSImageScaling

DECLARED IN AppKit/NSImageCell.h

SYNOPSIS

```
typedef enum {
    NSScaleProportionally,
    NSScaleToFit,
    NSScaleNone
} NSImageScaling;
```

DESCRIPTION This type defines the ways that an image can be scaled to fit an NSImageCell. The value NSScaleProportionally means that the image should be scaled in a way that preserves its proportions. The value NSScaleToFit means that the image should fit the NSView, even if that means its proportions must be distorted. The value NSScaleNone means that the image's size should be preserved, even if it must be clipped to fit the NSView.

NSInterfaceStyle

DECLARED IN AppKit/NSInterfaceStyle.h

SYNOPSIS

typedef enum {
    NSNoInterfaceStyle,
    NSNextStepInterfaceStyle,
    NSWindows95InterfaceStyle,
    NSMacintoshInterfaceStyle
} **NSInterfaceStyle**;

DECLARED IN This type defines the style of an application's user interface. It is returned by the **interfaceStyle** method and taken as an argument by the **setInterfaceStyle:** method. Both of the these methods are in the NSInterfaceStyle category of NSResponder.

For more information, see the function **NSInterfaceStyleForKey**.

NSLineBreakMode

DECLARED IN AppKit/NSParagraphStyle.h

SYNOPSIS

typedef enum _NSLineBreakMode {
    NSLineBreakByWordWrapping,
    NSLineBreakByCharWrapping,
    NSLineBreakByClipping,
    NSLineBreakByTruncatingHead,
    NSLineBreakByTruncatingTail,
    NSLineBreakByTruncatingMiddle
} **NSLineBreakMode**;

DESCRIPTION This type defines the ways that a long paragraph can be broken into lines. The possible values are described below.

Value	Meaning
NSLineBreakByWordWrapping	The default value. At the last possible word boundary, the paragraph wraps to the next line.

Value	Meaning
NSLineBreakByCharWrapping	At the last possible character, the paragraph wraps to the next line.
NSLineBreakByClipping	As much of the paragraph appears as will fit on a single line. This value has the same effect as NSLineBreakByTruncatingTail
NSLineBreakByTruncatingHead	As much of the paragraph appears as will fit on a single line. Characters from the start of the paragraph do not appear.
NSLineBreakByTruncatingTail	As much of the paragraph appears as will fit on a single line. Characters from the end of the paragraph do not appear.
NSLineBreakByTruncatingMiddle	As much of the paragraph appears as will fit on a single line. Characters from the middle of the paragraph do not appear.

NSLineMovementDirection

DECLARED IN AppKit/NSTextContainer.h

SYNOPSIS

```
typedef enum {
    NSLineDoesntMove,
    NSLineMovesLeft,
    NSLineMovesRight,
    NSLineMovesDown,
    NSLineMovesUp
} NSLineMovementDirection;
```

DESCRIPTION This is an argument type for the NSTextContainer method **lineFragmentRectForProposedRect:sweepDirection:movementDirection:remainingRect:**.

NSLineSweepDirection

DECLARED IN AppKit/NSTextContainer.h

SYNOPSIS

typedef enum {
    NSLineSweepLeft,
    NSLineSweepRight,
    NSLineSweepDown,
    NSLineSweepUp
} **NSLineSweepDirection**;

DESCRIPTION This is an argument type for the NSTextContainer method **lineFragmentRectForProposedRect:sweepDirection:movementDirection:remainingRect:**.

NSMatrixMode

DECLARED IN AppKit/NSMatrix.h

SYNOPSIS

typedef enum _NSMatrixMode {
    NSRadioModeMatrix,
    NSHighlightModeMatrix,
    NSListModeMatrix,
    NSTrackModeMatrix
} **NSMatrixMode**;

DESCRIPTION These constants represent the modes of operation of an NSMatrix, as described in the NSMatrix class specification.

NSModalSession

DECLARED IN AppKit/NSApplication.h

SYNOPSIS

typedef struct _NSModalSession ***NSModalSession**;

DESCRIPTION Variables of type NSModalSession point to information used by the system between **beginModalSession:for:** and **endModalSession:** messages.

NSPrinterTableStatus

DECLARED IN AppKit/NSPrinter.h

SYNOPSIS

typedef enum _NSPrinterTableStatus {
    NSPrinterTableOK,
    NSPrinterTableNotFound,
    NSPrinterTableError
} **NSPrinterTableStatus**;

DESCRIPTION These constants are used to describe the state of a printer-information table stored by an NSPrinter object. See the NSPrinter method **statusForTable:** for more information.

NSPrintingOrientation

DECLARED IN AppKit/NSPrintInfo.h

SYNOPSIS

typedef enum _NSPrintingOrientation {
    NSPortraitOrientation,
    NSLandscapeOrientation
} **NSPrintingOrientation**;

DESCRIPTION These constants represent the way a page is oriented for printing. In NSPortraitOrientation, the page is taller than it is wide; in NSLandscapeOrientation, the page is wider than it is tall. See the NSPrintInfo methods **orientation** and **setOrientation:** for more information.

NSPrintingPageOrder

DECLARED IN AppKit/NSPrintOperation.h

SYNOPSIS

typedef enum _NSPrintingPageOrder {
    NSDescendingPageOrder,
    NSSpecialPageOrder,
    NSAscendingPageOrder,
    NSUnknownPageOrder
} **NSPrintingPageOrder**;

DESCRIPTION This type represents the order in which pages are to be printed. The value NSSpecialPageOrder tells the spooler to not rearrange the pages. The value NSUnknownPageOrder means that no page order is written out. See the NSPrintOperation methods **pageOrder** and **setPageOrder:** for more information.

NSPrintingPaginationMode

DECLARED IN AppKit/NSPrintInfo.h

SYNOPSIS

typedef enum _NSPrintingPaginationMode {
    NSAutoPagination,
    NSFitPagination,
    NSClipPagination
} **NSPrintingPaginationMode**;

DESCRIPTION These constants represent the different ways in which an image is divided into pages. The value NSFitPagination forces the image to fit on one page. The value NSClipPagination allows the image to be clipped by the page.See the NSPrintInfo class specification for a fuller explanation.

NSRulerOrientation

DECLARED IN AppKit/NSRulerView.h

SYNOPSIS

```
typedef enum {
    NSHorizontalRuler,
    NSVerticalRuler
} NSRulerOrientation;
```

DESCRIPTION This type defines whether an NSRulerView will be displayed horizontally or vertically. It is the return type of NSRulerView's **orientation** method, and an argument to the NSRulerView methods **setOrientation:** and **initWithScrollView:orientation:**.

NSScrollArrowPosition

DECLARED IN AppKit/NSScroller.h

SYNOPSIS

```
typedef enum _NSScrollArrowPosition {
    NSScrollerArrowsMaxEnd,
    NSScrollerArrowsMinEnd,
    NSScrollerArrowsNone
} NSScrollArrowPosition;
```

DESCRIPTION These constants are used in NSScroller's **setArrowsPosition:** method to set the position of the arrows within the scroller.

NSScrollerArrow

DECLARED IN AppKit/NSScroller.h

SYNOPSIS

```
typedef enum _NSScrollerArrow {
    NSScrollerIncrementArrow,
    NSScrollerDecrementArrow
} NSScrollerArrow;
```

DESCRIPTION This is the type of the first argument to the NSScroller method **drawArrow:highlight:**. The value determines which scroll button is drawn.

NSScrollerPart

DECLARED IN AppKit/NSScroller.h

SYNOPSIS

```
typedef enum _NSScrollerPart {
    NSScrollerNoPart,
    NSScrollerDecrementPage,
    NSScrollerKnob,
    NSScrollerIncrementPage,
    NSScrollerDecrementLine,
    NSScrollerIncrementLine,
    NSScrollerKnobSlot
} NSScrollerPart;
```

DESCRIPTION These constants are used in Scroller's **hitPart** method to identify the part of the Scroller specified in a mouse event.

NSSelectionAffinity

DECLARED IN AppKit/NSTextView.h

SYNOPSIS

```
typedef enum _NSSelectionAffinity {
    NSSelectionAffinityUpstream,
    NSSelectionAffinityDownstream
} NSSelectionAffinity;
```

DESCRIPTION This is the return type of the NSTextView method **selectionAffinity**, and the type of the second argument to the NSTextView method **setSelectedRange:affinity:stillSelecting:**.

NSSelectionDirection

DECLARED IN AppKit/NSWindow.h

SYNOPSIS

typedef enum _NSSelectionDirection {
    NSDirectSelection,
    NSSelectingNext,
    NSSelectingPrevious
} **NSSelectionDirection**;

DESCRIPTION This is the return type of the NSWindow method **keyViewSelectionDirection**.

NSSelectionGranularity

DECLARED IN AppKit/NSTextView.h

SYNOPSIS

typedef enum _NSSelectionGranularity {
    NSSelectByCharacter,
    NSSelectByWord,
    NSSelectByParagraph
} **NSSelectionGranularity**;

DESCRIPTION This is the return type of the NSTextView method selectionGranularity, and the type of arguments to two other NSTextView methods, **setSelectionGranularity:** and **selectionRangeForProposedRange:granularity:**.

NSTextAlignment

DECLARED IN AppKit/NSText.h

SYNOPSIS

typedef enum _NSTextAlignment {
    NSLeftTextAlignment,
    NSRightTextAlignment,
    NSCenterTextAlignment,
    NSJustifiedTextAlignment,
    NSNaturalTextAlignment
} **NSTextAlignment**;

DESCRIPTION Variables of this type are used as arguments and return values for methods that specify text alignment.

NSTextTabType

DECLARED IN AppKit/NSParagraphStyle.h

SYNOPSIS

typedef enum _NSTextTabType {
    NSLeftTabStopType,
    NSRightTabStopType,
    NSCenterTabStopType,
    NSDecimalTabStopType
} **NSTextTabType**;

DESCRIPTION This is the return type of NSTextTab's **tabStopType** method, and an argument to NSTextTab's **initWithType:location:** method.

NSTIFFCompression

SYNOPSIS AppKit/NSBitmapImageRep.h

SYNOPSIS

typedef enum _NSTIFFCompression {
NSTIFFCompressionNone,
NSTIFFCompressionCCITTFAX3,
NSTIFFCompressionCCITTFAX4,
NSTIFFCompressionLZW,
NSTIFFCompressionJPEG,
NSTIFFCompressionNEXT,
NSTIFFCompressionPackBits,
NSTIFFCompressionOldJPEG
} **NSTIFFCompression**;

DESCRIPTION

These constants represent the various TIFF (*tag image file format*) data compression schemes. See the NSBitmapImageRep class specification for their meanings.

NSTitlePosition

DECLARED IN

AppKit/NSBox.h

SYNOPSIS

typedef enum _NSTitlePosition {
NSNoTitle,
NSAboveTop,
NSAtTop,
NSBelowTop,
NSAboveBottom,
NSAtBottom,
NSBelowBottom
} **NSTitlePosition**;

DESCRIPTION

This type represents the locations where an NSBox's title can be placed with respect to its border. Thus, for example, NSAboveTop means the title is above the top of the border, NSAtTop means the title breaks the top border, and so on. See the NSBox methods **titlePosition** and **setTitlePosition:**.

NSTrackingRectTag

DECLARED IN AppKit/NSView.h

SYNOPSIS typedef int **NSTrackingRectTag**;

DESCRIPTION This type describes the rectangle used to track the mouse. See the NSView methods **addTrackingRect:...** and **removeTrackingRect:**.

NSUsableScrollerParts

DECLARED IN AppKit/NSScroller.h

SYNOPSIS

```
typedef enum _NSUsableScrollerParts {
    NSNoScrollerParts,
    NSOnlyScrollerArrows,
    NSAllScrollerParts
} NSUsableScrollerParts;
```

DESCRIPTION This type defines the usable parts of an NSScroller; see the class specification for more information.

NSWindowDepth

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS typedef int **NSWindowDepth**;

DESCRIPTION This type represents the depth, or amount of memory, devoted to a single pixel in a window or screen.

Enumerations

NSApplication—Modal Session Return Values

DECLARED IN AppKit/NSApplication.h

SYNOPSIS

```
enum {
    NSRunStoppedResponse,
    NSRunAbortedResponse,
    NSRunContinuesResponse
};
```

DESCRIPTION Return values for **the NSApplication methods runModalFor:** and **runModalSession:**.

NSAttributedString—Underlining

DECLARED IN AppKit/NSAttributedString.h

SYNOPSIS

```
enum {
    NSSingleUnderlineStyle
};
```

DESCRIPTION This defines the only currently supported value for NSUnderlineStyleAttributeName.

NSButtonCell—State Masks

DECLARED IN AppKit/NSCell.h

SYNOPSIS

```
enum {
    NSNoCellMask,
    NSContentsCellMask,
    NSPushInCellMask,
    NSChangeGrayCellMask,
    NSChangeBackgroundCellMask
};
```

DESCRIPTION These masks are passed to the NSButtonCell methods **highlightsBy:** and **showsStateBy:**.

NSCell—Action Flags

DECLARED IN AppKit/NSEvent.h

SYNOPSIS

```
enum {
    NSLeftMouseDownMask,
    NSLeftMouseUpMask,
    NSRightMouseDownMask,
    NSRightMouseUpMask,
    NSMouseMovedMask,
    NSLeftMouseDraggedMask,
    NSRightMouseDraggedMask,
    NSMouseEnteredMask,
    NSMouseExitedMask,
    NSKeyDownMask,
    NSKeyUpMask,
    NSFlagsChangedMask,
    NSAppKitDefinedMask,
    NSSystemDefinedMask,
    NSApplicationDefinedMask,
    NSPeriodicMask,
    NSCursorUpdateMask,
    NSAnyEventMask
};
```

DESCRIPTION These constants are masks for different kinds of events. You pass them to NSCell's **sendActionOn:** method to indicate when an NSCell should send its action message.

NSCell—Data Entry Types

DECLARED IN AppKit/NSCell.h

SYNOPSIS

```
enum {
    NSAnyType,
    NSIntType,
    NSPositiveIntType,
    NSFloatType,
    NSPositiveFloatType,
    NSDoubleType,
    NSPositiveDoubleType
};
```

DESCRIPTION These constants represent the numeric data types that a text NSCell can accept. See NSCell's **setEntryType:** method for more information.

NSCell—States

DECLARED IN AppKit/NSCell.h

SYNOPSIS

```
enum {
    NSStateMixed,
    NSStateOff,
    NSStateOn
};
```

DESCRIPTION These constants are suggested parameter values for the NSCell method **setState:**.

NSColorPanel—Modes

DECLARED IN AppKit/NSColorPanel.h

SYNOPSIS

```
enum {
    NSGrayModeColorPanel,
    NSRGBModeColorPanel,
    NSCMYKModeColorPanel,
    NSHSBModeColorPanel,
    NSCustomPaletteModeColorPanel,
    NSColorListModeColorPanel,
    NSWheelModeColorPanel
};
```

DESCRIPTION These constants represent the possible modes of an NSColorPanel.

NSColorPanel—Mode Masks

DECLARED IN AppKit/NSColorPanel.h

SYNOPSIS

```
enum {
    NSColorPanelGrayModeMask,
    NSColorPanelRGBModeMask,
    NSColorPanelCMYKModeMask,
    NSColorPanelHSBModeMask,
    NSColorPanelCustomPaletteModeMask,
    NSColorPanelColorListModeMask,
    NSColorPanelWheelModeMask,
    NSColorPanelAllModesMask
};
```

DESCRIPTION These constants provide masks for the NSColorPanel modes.

NSDragging—Operations

DECLARED IN AppKit/NSDragging.h

SYNOPSIS

```
enum {
    NSDragOperationNone,
    NSDragOperationCopy,
    NSDragOperationLink,
    NSDragOperationGeneric,
    NSDragOperationPrivate,
    NSDragOperationAll
};
```

DESCRIPTION These constants define the operations that result from a user's drag. For full descriptions of their meanings and uses, see the method descriptions for **draggingSourceOperationMaskForLocal:** (in the NSDraggingSource protocol), **draggingSourceOperationMask** (in the NSDraggingInfo protocol), or **draggingEntered:** (in the NSDraggingDestination protocol).

NSEvent—Function-Key Unicodes

DECLARED IN AppKit/NSEvent.h

SYNOPSIS

```
enum {
    NSUpArrowFunctionKey = 0xF700,
    NSDownArrowFunctionKey = 0xF701,
    NSLeftArrowFunctionKey = 0xF702,
    NSRightArrowFunctionKey = 0xF703,
    NSF1FunctionKey = 0xF704,
    NSF2FunctionKey = 0xF705,
    NSF3FunctionKey = 0xF706,
    NSF4FunctionKey = 0xF707,
    NSF5FunctionKey = 0xF708,
    NSF6FunctionKey = 0xF709,
    NSF7FunctionKey = 0xF70A,
    NSF8FunctionKey = 0xF70B,
    NSF9FunctionKey = 0xF70C,
    NSF10FunctionKey = 0xF70D,
```

NSF11FunctionKey = 0xF70E,
NSF12FunctionKey = 0xF70F,
NSF13FunctionKey = 0xF710,
NSF14FunctionKey = 0xF711,
NSF15FunctionKey = 0xF712,
NSF16FunctionKey = 0xF713,
NSF17FunctionKey = 0xF714,
NSF18FunctionKey = 0xF715,
NSF19FunctionKey = 0xF716,
NSF20FunctionKey = 0xF717,
NSF21FunctionKey = 0xF718,
NSF22FunctionKey = 0xF719,
NSF23FunctionKey = 0xF71A,
NSF24FunctionKey = 0xF71B,
NSF25FunctionKey = 0xF71C,
NSF26FunctionKey = 0xF71D,
NSF27FunctionKey = 0xF71E,
NSF28FunctionKey = 0xF71F,
NSF29FunctionKey = 0xF720,
NSF30FunctionKey = 0xF721,
NSF31FunctionKey = 0xF722,
NSF32FunctionKey = 0xF723,
NSF33FunctionKey = 0xF724,
NSF34FunctionKey = 0xF725,
NSF35FunctionKey = 0xF726,
NSInsertFunctionKey = 0xF727,
NSDeleteFunctionKey = 0xF728,
NSHomeFunctionKey = 0xF729,
NSBeginFunctionKey = 0xF72A,
NSEndFunctionKey = 0xF72B,
NSPageUpFunctionKey = 0xF72C,
NSPageDownFunctionKey = 0xF72D,
NSPrintScreenFunctionKey = 0xF72E,
NSScrollLockFunctionKey = 0xF72F,
NSPauseFunctionKey = 0xF730,
NSSysReqFunctionKey = 0xF731,
NSBreakFunctionKey = 0xF732,
NSResetFunctionKey = 0xF733,
NSStopFunctionKey = 0xF734,
NSMenuFunctionKey = 0xF735,
NSUserFunctionKey = 0xF736,
NSSystemFunctionKey = 0xF737,
NSPrintFunctionKey = 0xF738,

NSClearLineFunctionKey = 0xF739,
NSClearDisplayFunctionKey = 0xF73A,
NSInsertLineFunctionKey = 0xF73B,
NSDeleteLineFunctionKey = 0xF73C,
NSInsertCharFunctionKey = 0xF73D,
NSDeleteCharFunctionKey = 0xF73E,
NSPrevFunctionKey = 0xF73F,
NSNextFunctionKey = 0xF740,
NSSelectFunctionKey = 0xF741,
NSExecuteFunctionKey = 0xF742,
NSUndoFunctionKey = 0xF743,
NSRedoFunctionKey = 0xF744,
NSFindFunctionKey = 0xF745,
NSHelpFunctionKey = 0xF746,
NSModeSwitchFunctionKey = 0xF747
};

DESCRIPTION These Unicodes (0xF700-0xF8FF) are reserved for function keys on the keyboard. Combined in NSStrings, they may be used in the return value of the NSEvent methods **characters...**, and in parameters of the NSEvent method **keyEventWithType:...characters:...**.

NSEvent—Modifier Flags

DECLARED IN AppKit/NSEvent.h

SYNOPSIS

enum {
NSAlphaShiftKeyMask,
NSShiftKeyMask,
NSControlKeyMask,
NSAlternateKeyMask,
NSCommandKeyMask,
NSNumericPadKeyMask,
NSHelpKeyMask,
NSFunctionKeyMask
};

DESCRIPTION These are device-independent bits found in event modifier flags.

NSEvent—Types Defined by the Application Kit

DECLARED IN AppKit/NSEvent.h

SYNOPSIS

```
enum {
    NSWindowExposedEventType,
    NSApplicationActivatedEventType,
    NSApplicationDeactivatedEventType,
    NSWindowMovedEventType,
    NSScreenChangedEventType
};
```

DESCRIPTION These constants represent the types of events defined by the Application Kit.

Constant	Meaning
NSWindowExposedEventType	A nonretained NSWindow has been exposed
NSApplicationActivatedEventType	The application has been activated
NSApplicationDeactivatedEventType	The application has been deactivated
NSWindowMovedEventType	An NSWindow has moved
NSScreenChangedEventType	An NSWindow has changed screens

NSEvent—Types Defined by the System

DECLARED IN AppKit/NSEvent.h

SYNOPSIS

```
enum {
    NSPowerOffEventType
};
```

DESCRIPTION This constant means that the user is turning off the computer.

NSFont—Traits

DECLARED IN AppKit/NSFontManager.h

SYNOPSIS

```
enum {
    NSItalicFontMask,
    NSBoldFontMask,
    NSUnboldFontMask,
    NSNonStandardCharacterSetFontMask,
    NSNarrowFontMask,
    NSExpandedFontMask,
    NSCondensedFontMask,
    NSSmallCapsFontMask,
    NSPosterFontMask,
    NSCompressedFontMask,
    NSFixedPitchFontMask,
    NSUnitalicFontMask
};
```

DESCRIPTION These constants are used by the NSFontManager to identify font traits. Some traits are mutually exclusive, such as NSExpandedFontMask and NSCondensedFontMask.

NSFontPanel—Tags for Subviews

DECLARED IN AppKit/NSFontPanel.h

SYNOPSIS

```
enum {
    NSFPPreviewButton,
    NSFPRevertButton,
    NSFPSetButton,
    NSFPPreviewField,
    NSFPSizeField,
    NSFPSizeTitle,
    NSFPCurrentField
};
```

DESCRIPTION These tags identify the NSViews within an NSFontPanel.

NSGlyph—Attributes

DECLARED IN AppKit/NSLayoutManager.h

SYNOPSIS

```
enum _NSGlyphAttribute {
    NSGlyphAttributeSoft,
    NSGlyphAttributeElastic,
    NSGlyphAttributeInscribe
};
```

DESCRIPTION These glyph attributes are used only inside the glyph generation machinery, but must be shared between components.

NSGlyph—Reserved Glyph Codes

DECLARED IN AppKit/NSFont.h

SYNOPSIS

```
enum {
    NSControlGlyph = 0x00FFFFFF,
    NSNullGlyph = 0x0
};
```

DESCRIPTION These two values are reserved for the two named NSGlyphs.

NSImageRep—Display Device Matching

DECLARED IN AppKit/NSImageRep.h

SYNOPSIS

```
enum {
    NSImageRepMatchesDevice
};
```

DESCRIPTION This constant is used by NSImageRep to indicate that the value of certain attributes, such as the number of colors, or bits-per-sample, will change to match the display device. See the NSImageRep class specification for more information.

NSPageLayoutPanel—Tags for Controls

DECLARED IN AppKit/NSPageLayout.h

SYNOPSIS

```
enum {
    NSPLImageButton,
    NSPLTitleField,
    NSPLPaperNameButton,
    NSPLUnitsButton,
    NSPLWidthForm,
    NSPLHeightForm,
    NSPLOrientationMatrix,
    NSPLCancelButton,
    NSPLOKButton
};
```

DESCRIPTION These constants represent the tag values of the controls displayed by an NSPageLayoutPanel.

NSPanel—Alert Panel Return Values

DECLARED IN AppKit/NSPanel.h

SYNOPSIS

```
enum {
    NSAlertDefaultReturn,
    NSAlertAlternateReturn,
    NSAlertOtherReturn,
    NSAlertErrorReturn
};
```

DESCRIPTION These constants define values returned by the **NSRunAlertPanel** function and by the NSApplication method **runModalSession:** when the modal session is run with an NSPanel provided by the **NSGetAlertPanel** function.

NSPanel—Modal Panel Return Values

DECLARED IN AppKit/NSPanel.h

SYNOPSIS

```
enum {
    NSOKButton,
    NSCancelButton
};
```

DESCRIPTION These are the possible return values for such methods as the **runModal...** methods of NSOpenPanel, which tells which button (OK or Cancel) the user has clicked on an open panel. For other uses of these return values, see the class descriptions for NSPageLayout, NSPrintPanel and NSSavePanel.

NSPrintPanel—Tags for Subviews

DECLARED IN AppKit/NSPrintPanel.h

SYNOPSIS

```
enum {
    NSPPSaveButton,
    NSPPPreviewButton,
    NSFaxButton,
    NSPPTitleField,
    NSPPImageButton,
    NSPPNameTitle,
    NSPPNameField,
    NSPPNoteTitle,
    NSPPNoteField,
    NSPPStatusTitle,
    NSPPStatusField,
    NSPPCopiesField,
    NSPPPageChoiceMatrix,
    NSPPPageRangeFrom,
    NSPPPageRangeTo,
    NSPPScaleField,
    NSPPOptionsButton,
    NSPPPaperFeedButton,
    NSPPLayoutButton
};
```

DESCRIPTION These constants define tags for identifying the NSViews in a print panel in environments other than Microsoft Windows. Windows has its own way of handling print panels.

NSRunLoop—Ordering Modes for NSApplication

DECLARED IN AppKit/NSApplication.h

SYNOPSIS

```
enum {
    NSUpdateWindowsRunLoopOrdering
};
```

DESCRIPTION This constants is used with NSRunLoop's **performSelector:target:argument:order:modes:** method.

NSRunLoop—Ordering Mode for NSDPSServerContext

DECLARED IN AppKit/NSDPSServerContext.h

SYNOPSIS

```
enum {
    DPSFlushContextRunLoopOrdering
};
```

DESCRIPTION This constants is used with NSRunLoop's method **performSelector:target:argument:order:modes:**.

NSRunLoop—Ordering Modes for NSWindow

DECLARED IN AppKit/NSWindow.h

SYNOPSIS

```
enum {
    NSDisplayWindowRunLoopOrdering,
    NSResetCursorRectsRunLoopOrdering
};
```

DESCRIPTION These constants are passed to NSRunLoop’s method **performSelector:target:argument:order:modes:**.

NSSavePanel—Tags for Subviews

DECLARED IN AppKit/NSSavePanel.h

SYNOPSIS

```
enum {
    NSFileHandlingPanelImageButton,
    NSFileHandlingPanelTitleField,
    NSFileHandlingPanelBrowser,
    NSFileHandlingPanelCancelButton,
    NSFileHandlingPanelOKButton,
    NSFileHandlingPanelForm,
    NSFileHandlingPanelHomeButton,
    NSFileHandlingPanelDiskButton,
    NSFileHandlingPanelDiskEjectButton
};
```

DESCRIPTION These constants define tags for identifying NSViews in an NSSavePanel.

NSTextAttachment—Attachment Character

DECLARED IN AppKit/NSTextAttachment.h

SYNOPSIS

```
enum {
    NSAttachmentCharacter = 0xfffc
};
```

DECLARED IN This Unicode indicates the presence of an attachment in an NSAttributedString. For more information, see the Class Cluster Description of NSAttributedStringAdditions.

NSText—Important Unicodes

DECLARED IN AppKit/NSText.h

SYNOPSIS

```
enum {
    NSParagraphSeparatorCharacter = 0x2029,
    NSLineSeparatorCharacter = 0x2028,
    NSTabCharacter = 0x0009,
    NSFormFeedCharacter = 0x000c,
    NSNewlineCharacter = 0x000a,
    NSCarriageReturnCharacter = 0x000d,
    NSEnterCharacter = 0x0003,
    NSBackspaceCharacter = 0x0008,
    NSBackTabCharacter = 0x0019,
    NSDeleteCharacter = 0x007f,
};
```

DESCRIPTION These constants specify several commonly-used Unicode characters.

NSText—Movement Codes

DECLARED IN AppKit/NSText.h

SYNOPSIS

```
enum {
    NSIllegalTextMovement,
    NSReturnTextMovement,
    NSTabTextMovement,
    NSBacktabTextMovement,
    NSLeftTextMovement,
    NSRightTextMovement,
    NSUpTextMovement,
    NSDownTextMovement
};
```

DESCRIPTION These constants are the codes for movement between fields. They are the possible int values for the NSTextMovement key of NSTextDidEndEditingNotification. For more information, see the "Notifications" section of the NSText class specification.

NSTextStorage—Editing

DECLARED IN AppKit/NSTextStorage.h

SYNOPSIS

```
enum {
    NSTextStorageEditedAttributes,
    NSTextStorageEditedCharacters
};
```

DESCRIPTION These values, which may be combined by a bitwise OR, help describe the changes that an editing session has made to an NSTextStorage object. They are the return values of the NSTextStorage method **editedMask**, and the parameter values for the second slot of the NSLayoutManager method **textStorage:edited:...**.

NSView—Resizing

DECLARED IN AppKit/NSView.h

SYNOPSIS

```
enum {
    NSViewNotSizable,
    NSViewMinXMargin,
    NSViewWidthSizable,
    NSViewMaxXMargin,
    NSViewMinYMargin,
    NSViewHeightSizable,
    NSViewMaxYMargin
};
```

DESCRIPTION Used to describe which parts of an NSView (or its margins) are resized when the NSView's superNSView is resized. See the NSView class specification for details.

NSWindow—Border Masks

DECLARED IN AppKit/NSWindow.h

SYNOPSIS

```
enum {
    NSBorderlessWindowMask,
    NSTitledWindowMask,
    NSClosableWindowMask,
    NSMiniaturizableWindowMask,
    NSResizableWindowMask
};
```

DESCRIPTION These determine the presence of a title and various buttons in an NSWindow's border.

NSWindow—Window Levels

DECLARED IN AppKit/NSWindow.h

SYNOPSIS

```
enum {
    NSNormalWindowLevel,
    NSFloatingWindowLevel,
    NSDockWindowLevel,
    NSSubmenuWindowLevel,
    NSTornOffMenuWindowLevel,
    NSMainMenuWindowLevel,
    NSModalPanelWindowLevel,
    NSPopUpMenuWindowLevel
};
```

DESCRIPTION These constants name the Application Kit's window levels. The stacking of levels takes precedence over the stacking of windows within each level. That is, even the bottom window in a level will obscure even the top window of the next level down.

The constant NSTornOffMenuWindowLevel is preferable to its synonym, NSSubmenuWindowLevel.

Global Variables

Application Kit—Exceptions

DECLARED IN AppKit/NSErrors.h

SYNOPSIS

NSString ***NSTextLineTooLongException**;
NSString ***NSTextNoSelectionException**;
NSString ***NSWordTablesWriteException**;
NSString ***NSWordTablesReadException**;
NSString ***NSTextReadException**;
NSString ***NSTextWriteException**;
NSString ***NSPasteboardCommunicationException**;
NSString ***NSPrintingCommunicationException**;
NSString ***NSAbortModalException**;
NSString ***NSAbortPrintingException**;
NSString ***NSIllegalSelectorException**;
NSString ***NSAppKitVirtualMemoryException**;
NSString ***NSBadRTFDirectiveException**;
NSString ***NSBadRTFFontTableException**;
NSString ***NSBadRTFStyleSheetException**;
NSString ***NSTypedStreamVersionException**;
NSString ***NSTIFFException**;
NSString ***NSPrintPackageException**;
NSString ***NSBadRTFColorTableException**;
NSString ***NSDraggingException**;

NSString ***NSColorListIOException**;
NSString ***NSColorListNotEditableException**;
NSString ***NSBadBitmapParametersException**;
NSString ***NSWindowServerCommunicationException**;
NSString ***NSFontUnavailableException**;
NSString ***NSPPDIncludeNotFoundException**;
NSString ***NSPPDParseException**;
NSString ***NSPPDIncludeStackOverflowException**;
NSString ***NSPPDIncludeStackUnderflowException**;
NSString ***NSRTFPropertyStackOverflowException**;
NSString ***NSAppKitIgnoredException**;
NSString ***NSBadComparisonException**;
NSString ***NSImageCacheException**;
NSString ***NSNibLoadingException**;
NSString ***NSBrowserIllegalDelegateException**;

DESCRIPTION These constants name the exceptions that the Application Kit can raise.

Display Device—Descriptions

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS
NSString ***NSDeviceResolution**;
NSString ***NSDeviceColorSpaceName**;
NSString ***NSDeviceBitsPerSample**;
NSString ***NSDeviceIsScreen**;
NSString ***NSDeviceIsPrinter**;
NSString ***NSDeviceSize**;

DESCRIPTION These are the keys for device description dictionaries, such as those returned by the **deviceDictionary** methods of NSPrinter, NSScreen and NSWindow.

NSDeviceResolution is an NSValue containing an NSSize in dots per inch. NSColorSpaceName is an NSString describing the color space of the device. NSDeviceBitsPerSample is an NSValue containing an int. NSDeviceIsScreen and NSDeviceIsPrinter are boolean values that tell whether the device is a screen or a printer. NSDeviceSize is an NSValue containing an NSSize that represents the device's size in points.

NSApplication—Notifications

DECLARED IN AppKit/NSApplication.h

SYNOPSIS

NSString ***NSApplicationDidBecomeActiveNotification**;
NSString ***NSApplicationDidFinishLaunchingNotification**;
NSString ***NSApplicationDidHideNotification**;
NSString ***NSApplicationDidResignActiveNotification**;
NSString ***NSApplicationDidUnhideNotification**;
NSString ***NSApplicationDidUpdateNotification**;
NSString ***NSApplicationWillBecomeActiveNotification**;
NSString ***NSApplicationWillFinishLaunchingNotification**;
NSString ***NSApplicationWillHideNotification**;
NSString ***NSApplicationWillResignActiveNotification**;
NSString ***NSApplicationWillUnhideNotification**;
NSString ***NSApplicationWillUpdateNotification**;
NSString ***NSApplicationWillTerminateNotification**;

DECLARED IN These are the notifications used with the methods of the NSApplicationNotifications category of NSObject.

NSApplication—Shared Application Object

DECLARED IN AppKit/NSApplication.h

SYNOPSIS id **NSApp**;

DESCRIPTION This variable designates the shared application object, created by NSApplication's **sharedApplication** method.

NSAttributedString—Attributes

DECLARED IN AppKit/NSAttributedString.h

SYNOPSIS

NSString ***NSFontAttributeName**;
NSString ***NSParagraphStyleAttributeName**;
NSString ***NSForegroundColorAttributeName**;
NSString ***NSUnderlineStyleAttributeName**;
NSString ***NSSuperscriptAttributeName**;
NSString ***NSBackgroundColorAttributeName**;
NSString ***NSAttachmentAttributeName**;
NSString ***NSLigatureAttributeName**;
NSString ***NSBaselineOffsetAttributeName**;
NSString ***NSKernAttributeName**;

DESCRIPTION These strings define the supported attributes of NSAttributedStrings. For more information, see the “Accessing Attributes” section in the NSAttributedString class cluster specification.

NSComboBox—Notifications

DECLARED IN AppKit/NSComboBox.h

SYNOPSIS

NSString ***NSComboBoxWillPopUpNotification**;
NSString ***NSComboBoxWillDismissNotification**;
NSString ***NSComboBoxSelectionDidChangeNotification**;
NSString ***NSComboBoxSelectionIsChangingNotification**;

DESCRIPTION These notifications are sent by NSComboBoxes.

NSColor—Color Space Names

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS

NSString ***NSCalibratedWhiteColorSpace**;
NSString ***NSCalibratedBlackColorSpace**;
NSString ***NSCalibratedRGBColorSpace**;
NSString ***NSDeviceWhiteColorSpace**;
NSString ***NSDeviceBlackColorSpace**;
NSString ***NSDeviceRGBColorSpace**;
NSString ***NSDeviceCMYKColorSpace**;
NSString ***NSNamedColorSpace**;
NSString ***NSCustomColorSpace**;

DESCRIPTION These are the predefined names for color spaces. In the two ...WhiteColorSpaces, white corresponds to a value of 1.0. In the two ...BlackColorSpaces, black corresponds to a value of 1.0. NSNamedColorSpace is used for “catalog” colors—that is, colors specified by names rather than coordinates. NSCustomColorSpace indicates a custom color space, which can be useful in working with images; unlike the other color spaces, NSCustomColorSpace is not used with NSColors.

NSColor—Grayscale Values

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS

const float **NSWhite**;
const float **NSLightGray**;
const float **NSDarkGray**;
const float **NSBlack**;

DESCRIPTION These are the standard gray values for the 2-bit deep grayscale color space.

NSColor—Notifications

DECLARED IN AppKit/NSColor.h

SYNOPSIS NSString ***NSSystemColorsDidChangeNotification**;

DESCRIPTION This notification is sent when the system colors have been changed (such as through a system control panel interface). For more on system colors, see the “System Colors” section of the NSColor class specification.

NSColorList—Notifications

DECLARED IN AppKit/NSColorList.h

SYNOPSIS NSString ***NSColorListDidChangeNotification**;

DESCRIPTION When an NSColorList changes, it posts this notification.

NSColorPanel—Notifications

DECLARED IN AppKit/NSColorPanel.h

SYNOPSIS NSString ***NSColorPanelColorDidChangeNotification**;

DESCRIPTION When an NSColorPanel changes, it posts this notification.

NSControl—Notifications

DECLARED IN AppKit/NSControl.h

SYNOPSIS NSString ***NSControlTextDidBeginEditingNotification**;
NSString ***NSControlTextDidEndEditingNotification**;
NSString ***NSControlTextDidChangeNotification**;

DESCRIPTION NSControls containing editable text can send these notifications. For more information, see the "Notifications" section of the NSControl class specification.

NSDataLink—Filename Extension

DECLARED IN AppKit/NSDataLink.h

SYNOPSIS NSString ***NSDataLinkFilenameExtension**;

DESCRIPTION NSDataLinkFilenameExtension is the filename extension used for links saved to files using NSDataLink's **saveLinkIn:** or **writeToFile:** methods.

NSFont—Keys to the AFM Dictionary

DECLARED IN AppKit/NSFont.h

SYNOPSIS

NSString ***NSAFMFamilyName**;
NSString ***NSAFMFontName**;
NSString ***NSAFMFormatVersion**;
NSString ***NSAFMFullName**;
NSString ***NSAFMNotice**;
NSString ***NSAFMVersion**;
NSString ***NSAFMWeight**;
NSString ***NSAFMEncodingScheme**;
NSString ***NSAFMCharacterSet**;
NSString ***NSAFMCapHeight**;
NSString ***NSAFMXHeight**;
NSString ***NSAFMAscender**;
NSString ***NSAFMDescender**;
NSString ***NSAFMUnderlinePosition**;
NSString ***NSAFMUnderlineThickness**;
NSString ***NSAFMItalicAngle**;
NSString ***NSAFMMappingScheme**;

DESCRIPTION These are the keys to the font information dictionary returned by NSFont's **afmDictionary** method. To convert values like NSAFMCapHeight to floats, use NSString's **floatValue** method.

For other font information, use NSFont's **afmFileContents** method.

NSFont—PostScript Transformation Matrix

DECLARED IN AppKit/NSFont.h

SYNOPSIS

const float ***NSFontIdentityMatrix**;

DESCRIPTION NSFontIdentityMatrix is a PostScript transformation matrix useful as a parameter to the NSFont method **fontWithName:matrix:**.

NSHelpManager—Notifications

DECLARED IN AppKit/NSHelpManager.h

SYNOPSIS NSString ***NSContextHelpModeDidActivateNotification**;
NSString ***NSContextHelpModeDidDeactivateNotification**;

DESCRIPTION These are notifications for the activation and deactivation of the context help mode.

NSImageRep—Notifications

DECLARED IN AppKit/NSImageRep.h

SYNOPSIS NSString ***NSImageRepRegistryDidChangeNotification**;

DESCRIPTION This notification is sent when the NSImageRep class registry changes.

NSInterfaceStyleDefault

DECLARED IN AppKit/NSInterfaceStyle.h

SYNOPSIS NSString ***NSInterfaceStyleDefault**;

DESCRIPTION NSInterfaceStyleDefault can be used to override the platform's default interface style. For more information, see the function **NSInterfaceStyleForKey**.

NSPasteboard—Names

DECLARED IN AppKit/NSPasteboard.h

SYNOPSIS
NSString ***NSGeneralPboard**;
NSString ***NSFontPboard**;
NSString ***NSRulerPboard**;
NSString ***NSFindPboard**;
NSString ***NSDragPboard**;

DESCRIPTION Some standard pasteboard names. See the NSPasteboard class specification for more information.

NSPasteboard—Type for Data Links

DECLARED IN AppKit/NSDataLink.h

SYNOPSIS NSString ***NSDataLinkPboardType**;

DESCRIPTION A pasteboard type for copying a data link to the pasteboard. See the NSDataLink class specification for more information.

NSPasteboard—Type for Selection Descriptions

DECLARED IN AppKit/NSSelection.h

SYNOPSIS NSString ***NSSelectionPboardType**;

DESCRIPTION A pasteboard type for copying selection descriptions to the pasteboard. See the NSSelection class specification for more information.

NSPasteboard—Types for Standard Data

DECLARED IN AppKit/NSPasteboard.h

SYNOPSIS

NSString ***NSStringPboardType**;
NSString ***NSFilenamesPboardType**;
NSString ***NSPostScriptPboardType**;
NSString ***NSTIFFPboardType**;
NSString ***NSRTFPboardType**;
NSString ***NSTabularTextPboardType**;
NSString ***NSFontPboardType**;
NSString ***NSRulerPboardType**;
NSString ***NSFileContentsPboardType**;
NSString ***NSColorPboardType**;
NSString ***NSRTFDPboardType**;

DESCRIPTION Some standard pasteboard data types. See the NSPasteboard class specification for more information.

NSPrintInfo—Dictionary Keys

DECLARED IN AppKit/NSPrintInfo.h

SYNOPSIS

NSString ***NSPrintPaperName**;
NSString ***NSPrintPaperSize**;
NSString ***NSPrintFormName**;
NSString ***NSPrintMustCollate**;
NSString ***NSPrintOrientation**;
NSString ***NSPrintLeftMargin**;
NSString ***NSPrintRightMargin**;
NSString ***NSPrintTopMargin**;
NSString ***NSPrintBottomMargin**;
NSString ***NSPrintHorizontallyCentered**;
NSString ***NSPrintVerticallyCentered**;
NSString ***NSPrintHorizontalPagination**;
NSString ***NSPrintVerticalPagination**;
NSString ***NSPrintScalingFactor**;
NSString ***NSPrintAllPages**;
NSString ***NSPrintReversePageOrder**;
NSString ***NSPrintFirstPage**;
NSString ***NSPrintLastPage**;
NSString ***NSPrintCopies**;
NSString ***NSPrintPagesPerSheet**;
NSString ***NSPrintJobFeatures**;
NSString ***NSPrintPaperFeed**;
NSString ***NSPrintManualFeed**;
NSString ***NSPrintPrinter**;
NSString ***NSPrintJobDisposition**;

NSString ***NSPrintSavePath**;
NSString ***NSPrintFaxReceiverNames**;
NSString ***NSPrintFaxReceiverNumbers**;
NSString ***NSPrintFaxSendTime**;
NSString ***NSPrintFaxUseCoverSheet**;
NSString ***NSPrintFaxCoverSheetName**;
NSString ***NSPrintFaxReturnReceipt**;
NSString ***NSPrintFaxHighResolution**;
NSString ***NSPrintFaxTrimPageEnds**;
NSString ***NSPrintFaxModem**;
NSString ***NSPrintSpoolJob**;
NSString ***NSPrintFaxJob**;
NSString ***NSPrintPreviewJob**;
NSString ***NSPrintSaveJob**;
NSString ***NSPrintCancelJob**;

DESCRIPTION These are the keys to the NSPrintInfo NSDictionary. For a table explaining them, see the NSPrintInfo method **initWithDictionary:**.

NSPopUpButton—Notification

DECLARED IN AppKit/NSPopUpButton.h

SYNOPSIS NSString ***NSPopUpButtonWillPopUpNotification**;

DESCRIPTION NSPopUpButton sends this notification when an instance of it is about to pop up.

NSPrintOperation—Exception

DECLARED IN AppKit/NSPrintOperation.h

SYNOPSIS NSString ***NSPrintOperationExistsException**;

DESCRIPTION This exception is raised when there is already a print operation in process. The methods that raise it are the **EPSOperation...** and **printOperation...** methods in NSPrintOperation:

NSRunLoop—Modes

DECLARED IN AppKit/NSApplication.h

SYNOPSIS NSString ***NSModalPanelRunLoopMode**;
NSString ***NSEventTrackingRunLoopMode**;

DESCRIPTION These are modes passed to NSRunLoop

NSSplitView—Notifications

DECLARED IN AppKit/NSSplitView.h

SYNOPSIS NSString ***NSSplitViewDidResizeSubviewsNotification**;
NSString ***NSSplitViewWillResizeSubviewsNotification**;

DESCRIPTION These are the notifications that an NSSplitView can send.

NSTableView—Notifications

DECLARED IN AppKit/NSTableView.h

SYNOPSIS NSString ***NSTableViewSelectionDidChangeNotification**;
NSString ***NSTableViewColumnDidMoveNotification**;
NSString ***NSTableViewColumnDidResizeNotification**;
NSString ***NSTableViewSelectionIsChangingNotification**;

DESCRIPTION These are the notifications that an NSTableView can send.

NSText—Notifications

DECLARED IN AppKit/NSText.h

SYNOPSIS NSString ***NSTextDidBeginEditingNotification**;
NSString ***NSTextDidEndEditingNotification**;
NSString ***NSTextDidChangeNotification**;

DESCRIPTION These notifications can be sent by an NSText object. For explanations, see the “Notifications” section of the NSText class specification.

NSTextStorage—Notifications

DECLARED IN AppKit/NSTextStorage.h

SYNOPSIS NSString ***NSTextStorageWillProcessEditingNotification**;
NSString ***NSTextStorageDidProcessEditingNotification**;

DESCRIPTION These notifications can be sent by an NSTextStorage object. For explanations, see the “Notifications” section of the NSTextStorage class specification.

NSTextView—Notifications

DECLARED IN AppKit/NSTextView.h

SYNOPSIS NSString ***NSTextViewWillChangeNotifyingTextViewNotification**;
NSString ***NSTextViewDidChangeSelectionNotification**;

DESCRIPTION These notifications can be sent by an NSTextView object. For explanations, see the “Notifications” section of the NSTextView class specification.

The notifications that NSTextView most often sends are the ones that it inherits from NSText.

NSView—Notifications

DECLARED IN AppKit/NSView.h

SYNOPSIS
NSString ***NSViewFrameDidChangeNotification**;
NSString ***NSViewFocusDidChangeNotification**;
NSString ***NSViewBoundsDidChangeNotification**;

DESCRIPTION These notifications are sent by NSViews.

The last notification, NSViewBoundsDidChangeNotification, is sent when the view bounds change but the frame does not. That is, it is sent whenever the view's bounds are translated, scaled or rotated, but not when the bounds change in response to, say, a **setFrameSize:** message.

NSWindow—Notifications

DECLARED IN AppKit/NSWindow.h

SYNOPSIS
NSString ***NSWindowDidBecomeKeyNotification**;
NSString ***NSWindowDidBecomeMainNotification**;
NSString ***NSWindowDidChangeScreenNotification**;
NSString ***NSWindowDidDeminiaturizeNotification**;
NSString ***NSWindowDidExposeNotification**;
NSString ***NSWindowDidMiniaturizeNotification**;
NSString ***NSWindowDidMoveNotification**;
NSString ***NSWindowDidResignKeyNotification**;
NSString ***NSWindowDidResignMainNotification**;
NSString ***NSWindowDidResizeNotification**;
NSString ***NSWindowDidUpdateNotification**;
NSString ***NSWindowWillCloseNotification**;
NSString ***NSWindowWillMiniaturizeNotification**;
NSString ***NSWindowWillMoveNotification**;

DESCRIPTION These are the notifications that can be sent by an NSWindow object. For explanations, see the “Notifications” section of the NSWindow class specification.

NSWindow—Sizes

DECLARED IN AppKit/NSWindow.h

SYNOPSIS

NSSize **NSIconSize**;
NSSize **NSTokenSize**;

DESCRIPTION On some platforms, a token is a bezeled tile used to represent a docked application or a miniaturized document, and an icon is the image drawn inside a token.

On platforms that support tokens and icons, these size constants can be used for drawing inside them. It is more portable, however, to change an icon by using the NSApplication method **setApplicationIconImage:** or the NSWindow method **setMiniwindowImage:**.

NSWorkspace—File Operation Constants

DECLARED IN AppKit/NSWorkspace.h

SYNOPSIS

NSString ***NSWorkspaceMoveOperation**;
NSString ***NSWorkspaceCopyOperation**;
NSString ***NSWorkspaceLinkOperation**;
NSString ***NSWorkspaceCompressOperation**;
NSString ***NSWorkspaceDecompressOperation**;
NSString ***NSWorkspaceEncryptOperation**;
NSString ***NSWorkspaceDecryptOperation**;
NSString ***NSWorkspaceDestroyOperation**;
NSString ***NSWorkspaceRecycleOperation**;
NSString ***NSWorkspaceDuplicateOperation**;

DESCRIPTION These constants define possible values for the *operation* slot in NSWorkspace's **performFileOperation:**... method.

NSWorkspace—File Types

DECLARED IN AppKit/NSWorkspace.h

SYNOPSIS
NSString ***NSPlainFileType**;
NSString ***NSDirectoryFileType**;
NSString ***NSApplicationFileType**;
NSString ***NSFilesystemFileType**;
NSString ***NSShellCommandFileType**;

DESCRIPTION These values are used in the final parameter slot of the NSWorkspace method **getInfoForFile: application:type:**.

NSWorkspace—Notifications

DECLARED IN AppKit/NSWorkspace.h

SYNOPSIS
NSString ***NSWorkspaceDidLaunchApplicationNotification**;
NSString ***NSWorkspaceDidMountNotification**;
NSString ***NSWorkspaceDidPerformFileOperationNotification**;
NSString ***NSWorkspaceDidTerminateApplicationNotification**;
NSString ***NSWorkspaceDidUnmountNotification**;
NSString ***NSWorkspaceWillLaunchApplicationNotification**;
NSString ***NSWorkspaceWillPowerOffNotification**;
NSString ***NSWorkspaceWillUnmountNotification**;

DESCRIPTION These notifications come through the special notification center. For more information, see the "Notifications" section of the NSWorkspace class specification.

NSChangeSpelling

Adopted By: NSText

Declared In: AppKit/NSSpellProtocol.h

Protocol Description

This protocol is implemented by objects in the responder chain that can correct a misspelled word. See the description of the NSSpellChecker class for more information.

Instance Methods

changeSpelling:

– (void)**changeSpelling:**(id)*sender*

Implement this method to replace the selected word in the receiver with a corrected version from the Spelling panel. This message is sent by the NSSpellChecker to the object whose text is being checked. To get the corrected spelling, ask the sender for the string value of its selected cell (visible to the user as the text field in the Spelling panel). This method should replace the selected portion of the text with the string that it gets from the NSSpellChecker.

NSColorPickingCustom

Adopted By: NSColorPicker

Declared In: AppKit/NSColorPicking.h

Protocol Description

Together with the NSColorPickingDefault protocol, NSColorPickingCustom provides a way to add color pickers—custom user interfaces for color selection—to an application's NSColorPanel. The NSColorPickingDefault protocol provides basic behavior for a color picker. The NSColorPicker class adopts the NSColorPickingDefault protocol. The easiest way to implement a color picker is to create a subclass of NSColorPicker and implement the NSColorPickingCustom protocol for this new class.

Note: All of NSColorPickingCustom's methods must be implemented by the custom color picker.

Here are the standard color picking modes and mode constants (defined in **AppKit/NSColorPanel.h**):

Mode	Color Mode Constant
Grayscale-Alpha	NSGrayModeColorPanel
Red-Green-Blue	NSRGBModeColorPanel
Cyan-Yellow-Magenta-Black	NSCMYKModeColorPanel
Hue-Saturation-Brightness	NSHSBModeColorPanel
Custom palette	NSCustomPaletteModeColorPanel
Custom color list	NSColorListModeColorPanel
Color wheel	NSWheelModeColorPanel

In grayscale-alpha, red-green-blue, cyan-magenta-yellow-black, and hue-saturation-brightness modes, the user adjusts colors by manipulating sliders. In the custom palette mode, the user can load an NSImage file (TIFF or EPS) into the NSColorPanel, then select colors from the image. In custom color list mode, the user can create and load lists of named colors. The two custom modes provide NSPopUpLists for loading and saving files. Finally, color wheel mode provides a simplified control for selecting colors.

If your color picker includes submodes, you should define a unique value for each submode. As an example, the slider picker has four values defined in the above list (NSGrayModeColorPanel,

NSRGBModeColorPanel, NSCMYKModeColorPanel, and NSHSBModeColorPanel)—one for each of its submodes.

Method Types

Setting the Current Color

– setColor:

Getting the Mode

– currentMode
– supportsMode:

Getting the View

– provideNewView:

Instance Methods

currentMode

– (int)**currentMode**

Returns the color picker's current mode (or submode, if applicable). The returned value should be unique to your color picker. See this protocol description's list of the unique values for the standard color pickers used by the Application Kit.

See also: **– supportsMode:**

provideNewView:

– (NSView *)**provideNewView:**(BOOL)*initialRequest*

Returns the view containing the color picker's user interface. This message is sent to the color picker whenever the color panel attempts to display it. This may be when the panel is first presented, when the user switches pickers, or when the picker is switched through API. The argument *initialRequest* is YES only when this method is first invoked for your color picker. If *initialRequest* is YES, the method should perform any initialization required (such as lazily loading a nib file, initializing the view, or performing any other custom initialization required for your picker). The NSView returned by this method should be set to automatically resize both its width and height.

setColor:

– (void)**setColor:**(NSColor *)*color*

Adjusts the color picker to make *color* the currently selected color. This method is invoked on the current color picker each time NSColorPanel's **setColor:** method is invoked. If *color* is actually different from the color picker's color (as it would be if, for example, the user dragged a color into NSColorPanel's color well), this method could be used to update the color picker's color to reflect the change.

supportsMode:

– (BOOL)**supportsMode:**(int)*mode*

Returns whether or not the receiver supports the specified picking mode. This method is invoked when the NSColorPanel's is first initialized: It is used to attempt to restore the user's previously selected mode. It is also invoked by NSColorPanel's **setMode:** method to find the color picker that supports a particular mode. See this protocol description's list of the unique mode values for the standard color pickers used by the Application Kit.

See also: **– currentMode**

NSColorPickingDefault

Adopted By: NSColorPicker

Declared In: AppKit/NSColorPicking.h

Protocol Description

The NSColorPickingDefault protocol, together with the NSColorPickingCustom protocol, provides an interface for adding color pickers—custom user interfaces for color selection—to an application's NSColorPanel. The NSColorPickingDefault protocol provides basic behavior for a color picker. The NSColorPickingCustom protocol provides implementation-specific behavior.

The NSColorPicker class implements the NSColorPickingDefault protocol. The simplest way to implement your own color picker is to create a subclass of NSColorPicker, implementing the NSColorPickingCustom protocol for that subclass. However, it's possible to create a subclass of another class, such as NSView, and use it as a base upon which to add the methods of both NSColorPickingDefault and NSColorPickingCustom.

Color Picker Bundles

A class that implements the NSColorPickingDefault and NSColorPickingCustom protocols needs to be compiled and linked in an application's object file. However, your application need not explicitly create an instance of this class. Instead, your application's file package should include a directory named **ColorPickers**; within this directory you should place a directory *MyPickerClass***.bundle** for each custom color picker your application implements. This bundle should contain all resources required for your color picker: nib files, TIFF files, and so on.

NSColorPanel will allocate and initialize an instance of each class for which a bundle is found in the **ColorPickers** directory. The class name is assumed to be the bundle directory name minus the **.bundle** extension.

Color Picker Buttons

NSColorPanel lets the user select a color picker from an NSMatrix of NSButtonCells. This protocol includes methods for providing and manipulating the image that gets displayed on the button.

Color Mask and Color Modes

The color mask determines which color mode is enabled for NSColorPanel. This mask is set before you initialize a new instance of NSColorPanel. NSColorPanelAllModesMask represents the logical OR of the

other color mask constants: It causes the NSColorPanel to display all standard color pickers. When initializing a new instance of NSColorPanel, you can logically OR any combination of color mask constants to restrict the available color modes. The predefined color mask constants are:

Mode	Color Mask Constant
Grayscale-Alpha	NSColorPanelGrayModeMask
Red-Green-Blue	NSColorPanelRGBModeMask
Cyan-Yellow-Magenta-Black	NSColorPanelCMYKModeMask
Hue-Saturation-Brightness	NSColorPanelHSBModeMask
Custom palette	NSColorPanelCustomPaletteModeMask
Custom color list	NSColorPanelColorListModeMask
Color wheel	NSColorPanelWheelModeMask
All of the above	NSColorPanelAllModesMask

When an application's instance of NSColorPanel is masked for more than one color mode, your program can set its active mode by invoking the **setMode:** method with a color mode constant as its argument; the user can set the mode by clicking buttons on the panel. Here are the standard color modes and mode constants:

Mode	Color Mode Constant
Grayscale-Alpha	NSGrayModeColorPanel
Red-Green-Blue	NSRGBModeColorPanel
Cyan-Yellow-Magenta-Black	NSCMYKModeColorPanel
Hue-Saturation-Brightness	NSHSBModeColorPanel
Custom palette	NSCustomPaletteModeColorPanel
Custom color list	NSColorListModeColorPanel
Color wheel	NSWheelModeColorPanel

In grayscale-alpha, red-green-blue, cyan-magenta-yellow-black, and hue-saturation-brightness modes, the user adjusts colors by manipulating sliders. In the custom palette mode, the user can load an NSImage file (TIFF or EPS) into the NSColorPanel, then select colors from the image. In custom color list mode, the user can create and load lists of named colors. The two custom modes provide NSPopUpLists for loading and saving files. Finally, color wheel mode provides a simplified control for selecting colors.

These constants are defined in **AppKit/NSColorPanel.h**.

Method Types

Initializing a Color Picker

– initWithPickerMask:colorPanel:

Setting the Mode

– setMode:

Using Color Lists

– attachColorList:
– detachColorList:

Adding Button Images

– insertNewButtonImage:in:
– provideNewButtonImage

Showing Opacity Controls

– alphaControlAddedOrRemoved:

Responding to a Resized View

– viewSizeChanged:

Instance Methods

alphaControlAddedOrRemoved:

– (void)**alphaControlAddedOrRemoved:**(id)*sender*

Sent by the color panel when the opacity controls have been hidden or displayed. Invoked automatically when the NSColorPanel's opacity slider is added or removed; you never invoke this method directly.

If the color picker has its own opacity controls, it should hide or display them, depending on whether the sender's **showsAlpha** method returns NO or YES.

attachColorList:

– (void)**attachColorList:**(NSColorList *)*colorList*

Tells the color picker to attach the given *colorList*, if it isn't already displaying the list. You never invoke this method; it's invoked automatically by the NSColorPanel when its **attachColorList:** method is invoked. Since NSColorPanel's list mode manages NSColorLists, this method need only be implemented by a custom color picker that manages NSColorLists itself. This method ordinarily doesn't do anything, since NSColorPanel's list mode manages NSColorLists.

See also: **– detachColorList:**

detachColorList:

– (void)**detachColorList:**(NSColorList *)*colorList*

Tells the color picker to detach the given *colorList*, unless the receiver isn't displaying the list. You never invoke this method; it's invoked automatically by the NSColorPanel when its **detachColorList:** method is invoked. Since NSColorPanel's list mode manages NSColorLists, this method need only be implemented by a custom color picker that manages NSColorLists itself. This method ordinarily doesn't do anything, since NSColorPanel's list mode manages NSColorLists.

See also: **– attachColorList:**

initWithPickerMask:colorPanel:

– (id)**initWithPickerMask:**(int)*mask*
colorPanel:(NSColorPanel *)*owningColorPanel*

Notifies the color picker of the color panel's mask and initializes the color picker. This method is sent by the NSColorPanel to all implementors of the color picking protocols when the application's color panel is first initialized. In order for your color picker to receive this message, it must have a bundle in your application's "ColorPickers" directory (described in "Color Picker Bundles" in the Protocol Description).

mask is determined by the argument to the NSColorPanel method **setPickerMask:**. If no mask has been set, *mask* is NSColorPanelAllModesMask. If your color picker supports any additional modes, you should invoke the **setPickerMask:** method when your application initializes to notify the NSColorPanel class. The standard mask constants are:

Mode	Color Mask Constant
Grayscale-Alpha	NSColorPanelGrayModeMask
Red-Green-Blue	NSColorPanelRGBModeMask

Mode	Color Mask Constant
Cyan-Yellow-Magenta-Black	NSColorPanelCMYKModeMask
Hue-Saturation-Brightness	NSColorPanelHSBModeMask
Custom palette	NSColorPanelCustomPaletteModeMask
Custom color list	NSColorPanelColorListModeMask
Color wheel	NSColorPanelWheelModeMask
All of the above	NSColorPanelAllModesMask

This method should examine the mask and determine whether it supports any of the modes included there. You may also check the value in *mask* to enable or disable any subpickers or optional controls implemented by your color picker. Your color picker may also retain *owningColorPanel* in an instance variable for future communication with the color panel.

This method is provided to initialize your color picker; however, much of a color picker's initialization may be done lazily through the NSColorPickingCustom protocol's **provideNewView:** method. If your color picker responds to any of the modes represented in *mask*, it should perform its initialization and return **self**. Color pickers that do so have their buttons inserted in the color panel and continue to receive messages from the panel as the user manipulates it. If the color picker doesn't respond to any of the modes represented in mask, it should do nothing and return **nil**.

See also: **+ setPickerMask:** (NSColorPanel class)

insertNewButtonImage:in:

– (void)**insertNewButtonImage:**(NSImage *)*newButtonImage*
in:(NSButtonCell *)*buttonCell*

Sets *newButtonImage* as *buttonCell*'s image. *buttonCell* is the NSButtonCell object that lets the user choose the picker from the color panel—the color picker's representation in the NSColorPanel's picker NSMatrix. This method should perform application-specific manipulation of the image before it's inserted and displayed by the button cell.

See also: **– provideNewButtonImage**

provideNewButtonImage

– (NSImage *)**provideNewButtonImage**

Returns the image for the mode button that the user uses to select this picker in the color panel, that is, the color picker's representation in the NSColorPanel's picker NSMatrix. (This is the same image that the color panel uses as an argument when sending the **insertNewButtonImage:in:** message.)

setMode:

– (void)**setMode:**(int)*mode*

Sets the color picker's mode. This method is invoked by NSColorPanel's **setMode:** method to ensure that the color picker reflects the current mode. For example, invoke this method during color picker initialization to ensure that all color pickers are restored to the mode the user left them in the last time an NSColorPanel was used.

Most color pickers have only one mode, and thus don't need to do any work in this method. An example of a color picker that uses this method is the slider picker, which can choose from one of several submodes depending on the value of *mode*. The available modes are:

Mode	Color Mode Constant
Grayscale-Alpha	NSGrayModeColorPanel
Red-Green-Blue	NSRGBModeColorPanel
Cyan-Yellow-Magenta-Black	NSCMYKModeColorPanel
Hue-Saturation-Brightness	NSHSBModeColorPanel
Custom palette	NSCustomPaletteModeColorPanel
Custom color list	NSColorListModeColorPanel
Color wheel	NSWheelModeColorPanel

viewSizeChanged:

– (void)**viewSizeChanged:**(id)*sender*

Tells the color picker when the NSColorPanel's view size changes in a way that might affect the color picker. *sender* is the NSColorPanel that contains the color picker. Use this method to perform special preparation when resizing the color picker's view. Since this method is invoked only as appropriate, it's

better to implement this method than to override the method **superviewSizeChanged:** for the NSView in which the color picker's user interface is contained.

See also: **– provideNewView:** (NSColorPickingCustom protocol)

NSComboBoxCellDataSource

(Informal Protocol)

Category Of: NSObject

Declared In: AppKit/NSComboBoxCell.h

Category Description

The NSComboBoxCellDataSource category declares the methods that an NSComboBoxCell uses to access the contents of its data source object. The combo box cell determines how many items to display by sending a **numberOfItemsInComboBoxCell:** message, and accesses individual values with the **comboBoxCell: objectValueForItemAtIndex:** method. Incremental searches—performed when a user types into the combo box's text field while the pop-up list is displayed—are performed by sending **comboBoxCell: indexOfItemWithStringValue:** messages to the combo box cell's data source.

The NSComboBoxCell treats objects provided by its data source as values to be displayed in the combo box's pop-up list. If these objects aren't of common value classes—such as NSString, NSNumber, and so on—you'll need to create a custom NSFormatter to display them. See the NSFormatter class specification for more information.

When an NSComboBoxCellDataSource is asked to supply a data item, the NSComboBoxCell that sends the request is provided as a parameter. This allows a single data source object to manage several sets of data, choosing the appropriate set based on the identify of the NSComboBoxCell that sends the message.

Instance Methods

comboBoxCell:indexOfItemWithStringValue:

– (unsigned int)**comboBoxCell:**(NSComboBoxCell *)*aComboBoxCell*
indexOfItemWithStringValue:(NSString *)*aString*

An NSComboBoxCell uses this method to perform incremental—or "smart"—searched when the user types into the text field with the pop-up list displayed. Your implementation of this method should return the index for the item which matches *aString*, or NSNotFound if no item matches. This method is optional; if you don't provide an implementation for this method, no searches occur.

comboBoxCell:objectValueForItemAtIndex:

– (id)**comboBoxCell:**(NSComboBoxCell *)*aComboBoxCell* **objectValueForItemAtIndex:**(int)*index*

Implement this method to return the object that corresponds to the item at *index* in *aComboBoxCell*. Your data source must implement this method.

numberOfItemsInComboBoxCell:

– (int)**numberOfItemsInComboBoxCell:**(NSComboBoxCell *)*aComboBoxCell*

Implement this method to return the number of items managed for *aComboBoxCell* by your data source object. An NSComboBoxCell uses this method to determine how many items it should display in its pop-up list. Your data source must implement this method.

NSComboBoxDataSource

(Informal Protocol)

Category Of: NSObject

Declared In: AppKit/NSComboBox.h

Category Description

The NSComboBoxDataSource category declares the methods that an NSComboBox uses to access the contents of its data source object. The combo box determines how many items to display by sending a **numberOfItemsInComboBox:** message, and accesses individual values with the **comboBox: objectValueForItemAtIndex:** method. Incremental searches—performed when a user types into the combo box's text field while the pop-up list is displayed—are performed by sending **comboBox: indexOfItemWithStringValue:** messages to the combo box's data source.

The NSComboBox treats objects provided by its data source as values to be displayed in the combo box's pop-up list. If these objects aren't of common value classes—such as NSString, NSNumber, and so on—you'll need to create a custom NSFormatter to display them. See the NSFormatter class specification for more information.

When an NSComboBoxDataSource is asked to supply a data item, the NSComboBox that sends the request is provided as a parameter. This allows a single data source object to manage several sets of data, choosing the appropriate set based on the identify of the NSComboBox that sends the message.

Instance Methods

comboBox:indexOfItemWithStringValue:

– (unsigned int)**comboBox:**(NSComboBox *)*aComboBox* **indexOfItemWithStringValue:** (NSString *)*aString*

An NSComboBox uses this method to perform incremental—or "smart"—searched when the user types into the text field with the pop-up list displayed. Your implementation of this method should return the index for the item which matches *aString*, or NSNotFound if no item matches. This method is optional; if you don't provide an implementation for this method, no searches occur.

comboBox:objectValueForItemAtIndex:

– (id)**comboBox:**(NSComboBox *)*aComboBox* **objectValueForItemAtIndex:**(int)*index*

Implement this method to return the object that corresponds to the item at *index* in *aComboBox*. Your data source must implement this method.

numberOfItemsInComboBox:

– (int)**numberOfItemsInComboBox:**(NSComboBox *)*aComboBox*

Implement this method to return the number of items managed for *aComboBox* by your data source object. An NSComboBox uses this method to determine how many items it should display in its pop-up list. Your data source must implement this method.

NSDPSContextNotification

Adopted By: no OpenStep classes

Declared In: AppKit/NSDPSContext.h

Protocol Description

The NSDPSContextNotification protocol supplies information about the execution status of a sequence of PostScript commands previously sent to the Display PostScript server.

Instance Methods

contextFinishedExecuting:

– (void)**contextFinishedExecuting:**(NSDPSContext *)*context*

Notifies the receiver that the context has finished executing a batch of PostScript commands. See **notifyObjectWhenFinishedExecuting:** (NSDPSContext).

NSDraggingDestination

(informal protocol)

Category Of: NSObject

Declared In: AppKit/NSDragging.h

Description

The NSDraggingDestination informal protocol declares methods that the destination (or recipient) of a dragged image must implement. The destination automatically receives NSDraggingDestination messages as an image enters, moves around inside, and then exits or is released within the destination's boundaries.

In the text here and in the other dragging protocol descriptions, the term *dragging session* is the entire process during which an image is selected, dragged, released, and absorbed or rejected by the destination. A *dragging operation* is the action that the destination takes in absorbing the image when it's released. The *dragging source* is the object that "owns" the image that's being dragged. It's specified as an argument to the **dragImage:at:offset:event:pasteboard:source:slideBack:** message, sent to a window or view object, that instigated the dragging session.

The Dragged Image

The image that's dragged in an image-dragging session is simply an image that represents data that resides on the pasteboard. Although a dragging destination can access the image (through the **draggedImage** method described in the NSDraggingInfo protocol), its primary concern is with the pasteboard data that the image represents—the dragging operation that a destination ultimately performs is on the pasteboard data, not on the image itself.

Valid Destinations

Dragging is a visual phenomenon. To be an image-dragging destination, an object must represent a portion of screen real estate; thus, only window and view objects can be destinations. Furthermore, you must register the pasteboard types that the object will accept by sending the object a **registerForDraggedTypes:** message, defined in both NSWindow and NSView. During a dragging session, a candidate destination only receives NSDraggingDestination messages if the destination is registered for a pasteboard type that matches the type of the pasteboard data being dragged. See the NSPasteboard class specification for more information about pasteboard types.

Although NSDraggingDestination is declared as an informal protocol, the NSWindow and NSView subclasses that you create to adopt the protocol need only implement those methods that are pertinent. (The NSWindow and NSView classes provide private implementations for all of the methods.) Either a window

object or its delegate may implement these methods; however, the delegate's implementation takes precedence if there are implementations in both places.

The Sender of Destination Messages

Each of the NSDraggingDestination methods sports a single argument: *sender*, the object that invoked the method. Within its implementations of the NSDraggingDestination methods, the destination can send NSDraggingInfo protocol messages to *sender* to get more information on the current dragging session.

The Order of Destination Messages

The six NSDraggingDestination methods are invoked in a distinct order:

- As the image is dragged into the destination's boundaries, the destination is sent a **draggingEntered:** message
- While the image remains within the destination, a series of **draggingUpdated:** messages are sent.
- If the image is dragged out of the destination, **draggingExited:** is sent and the sequence of NSDraggingDestination messages stops. If it re-enters, the sequence begins again (with a new **draggingEntered:** message).
- When the image is released, it either slides back to its source (and breaks the sequence) or a **prepareForDragOperation:** message is sent to the destination, depending on the value returned by the most recent invocation of **draggingEntered:** or **draggingUpdated:**.
- If the **prepareForDragOperation:** message returned YES, a **performDragOperation:** message is sent.
- Finally, if **performDragOperation:** returned YES, **concludeDragOperation:** is sent.

Method Types

Before the image is released

– draggingEntered:
– draggingUpdated:
– draggingExited:

After the image is released

– prepareForDragOperation:
– performDragOperation:
– concludeDragOperation:

Instance Methods

concludeDragOperation:

– (void)**concludeDragOperation:**(id <NSDraggingInfo>)*sender*

Invoked when the dragging operation is complete and the previous **performDragOperation:** returned YES. The destination implements this method to perform any tidying up that it needs to do, such as updating its visual representation now that it has incorporated the dragged data. This is the last message that's sent from *sender* to the destination during a dragging session.

draggingEntered:

– (unsigned int)**draggingEntered:**(id <NSDraggingInfo>)*sender*

Invoked when a dragged image enters the destination. Specifically, this method is invoked when the mouse pointer enters the destination's bounds rectangle (if it's a view object) or its frame rectangle (if it's a window object).

This method must return a value that indicates which dragging operation the destination will perform when the image is released. In deciding which dragging operation to return, the method should evaluate the overlap between both the dragging operations allowed by the source (accessible through the **draggingSourceOperationMask** method) and the dragging operations and pasteboard data types that the destination itself supports. The returned value should be exactly one of the following:

Option	Meaning
NSDragOperationCopy	The data represented by the image will be copied.
NSDragOperationLink	The data will be shared.
NSDragOperationGeneric	The operation will be defined by the destination.
NSDragOperationPrivate	The operation is negotiated privately between the source and the destination.
NSDragOperationAll	Combines all the above.

If none of the operations is appropriate, this method should return NSDragOperationNone (this is the default response if the method isn't implemented by the destination).

The code below is a simple example of a method that responds distinctly when one of two different types of data is dragged into the destination view or window. If the dragged data is a color and the source object permits copying, the return value indicates that the destination will permit copying of the color data on the pasteboard. If the dragged data is an RTF file and the source object permits linking, the return value

indicates that the destination will permit linking of the RTF file on the pasteboard. Otherwise the code returns NSDragOperationNone, indicating that the destination will not permit any dragging operations with the data on pasteboard.

```
- (unsigned int)draggingEntered:(id <NSDraggingInfo>)sender
{
    NSPasteboard *pboard;
    NSDragOperation sourceDragMask;

    sourceDragMask = [sender draggingSourceOperationMask];
    pboard = [sender draggingPasteboard];

    if ([[pboard types] indexOfObject:NSColorPboardType] != NSNotFound) {
        if (sourceDragMask & NSDragOperationCopy) {
            return NSDragOperationCopy;
        }
    }
    if ([[pboard types] indexOfObject:NSRTFPboardType] != NSNotFound) {
        if (sourceDragMask & NSDragOperationLink) {
            return NSDragOperationLink;
        }
    }
    return NSDragOperationNone;
}
```

See also: – **draggingUpdated:**, – **draggingExited:**, – **prepareForDragOperation:**

draggingExited:

– (void)**draggingExited:**(id <NSDraggingInfo>)*sender*

Invoked when the dragged image exits the destination's bounds rectangle (in the case of a view object) or its frame rectangle (in the case of a window object).

draggingUpdated:

– (unsigned int)**draggingUpdated:**(id <NSDraggingInfo>)*sender*

Invoked periodically as the image is held within the destination. The messages continue until the image is either released or dragged out of the window or view. The return value should be one of the dragging operation options listed under the **draggingEntered:** method. The default return value (if this method isn't implemented by the destination) is the value returned by the previous **draggingEntered:** message.

This method provides the destination with an opportunity to modify the dragging operation depending on the position of the mouse pointer inside of the destination view or window object. For example, you may have several graphics or areas of text contained within the same view and wish to tailor the dragging

operation, or to ignore the drag event completely, depending upon which object is underneath the mouse pointer at the time when the user releases the dragged image and the **performDragOperation:** method is invoked.

You typically examine the contents of the pasteboard in the **draggingEntered:** method, where this examination is performed only once, rather than in the **draggingUpdated:** method, which is invoked multiple times.

Only one destination at a time receives a sequence of **draggingUpdated:** messages. If the mouse pointer is within the bounds of two overlapping views that are both valid destinations, the uppermost view receives these messages until the image is either released or dragged out.

See also: – **draggingExited:**, – **prepareForDragOperation:**

performDragOperation:

– (BOOL)**performDragOperation:**(id <NSDraggingInfo>)*sender*

Invoked after the released image has been removed from the screen and the previous **prepareForDragOperation:** message has returned YES. The destination should implement this method to do the real work of importing the pasteboard data represented by the image. If the destination accepts the data, it returns YES, otherwise it returns NO. The default is to return NO.

See also: – **concludeDragOperation:**

prepareForDragOperation:

– (BOOL)**prepareForDragOperation:**(id <NSDraggingInfo>)*sender*

Invoked when the image is released, if the most recent **draggingEntered:** or **draggingUpdated:** message returned an acceptable drag-operation value. Returns YES if the receiver agrees to perform the drag operation and NO if not.

See also: – **performDragOperation:**

NSDraggingInfo

Adopted By: no Rhapsody classes

Declared In: AppKit/NSDragging.h

Protocol Description

The NSDraggingInfo protocol declares methods that supply information about a *dragging session* (see the NSDraggingDestination protocol for definitions of dragging terms). NSDraggingInfo methods are designed to be invoked from within a class's implementation of NSDraggingDestination informal protocol methods. The Application Kit automatically passes an object that conforms to the NSDraggingInfo protocol as the argument to each of the methods defined by NSDraggingDestination. NSDraggingInfo messages should be sent to this object; you never need to create a class that implements the NSDraggingInfo protocol.

Method Types

Dragging-session information

– draggingSource
– draggingSourceOperationMask
– draggingDestinationWindow
– draggingPasteboard
– draggingSequenceNumber
– draggingLocation

Image information

– draggedImage
– draggedImageLocation

Sliding the image

– slideDraggedImageTo:

Instance Methods

draggedImage

– (NSImage *)**draggedImage**

Returns the image being dragged. This image object visually represents the data put on the pasteboard during the drag operation; however, it is the pasteboard data and not this image that are ultimately utilized in the dragging operation.

See also: – **draggedImageLocation**

draggedImageLocation

– (NSPoint)**draggedImageLocation**

Returns the current location of the dragged image's origin in the base coordinate system of the destination object's window. The image moves along with the mouse pointer (the position of which is given by **draggingLocation**) but may be positioned at some offset.

See also: – **draggedImage**

draggingDestinationWindow

– (NSWindow *)**draggingDestinationWindow**

Returns the destination window for the dragging operation. Either this window is the destination itself, or it contains the view object that is the destination.

draggingLocation

– (NSPoint)**draggingLocation**

Returns the current location of the mouse pointer in the base coordinate system of the destination object's window.

See also: – **draggedImageLocation**

draggingPasteboard

– (NSPasteboard *)**draggingPasteboard**

Returns the pasteboard object that holds the data being dragged. The dragging operation that is ultimately performed utilizes this pasteboard data and not the image returned by the **draggedImage** method.

draggingSequenceNumber

– (int)**draggingSequenceNumber**

Returns a number that uniquely identifies the dragging session.

draggingSource

– (id)**draggingSource**

Returns the source, or owner, of the dragged data or **nil** if the source isn't in the same application as the destination. The dragging source implements methods from the NSDraggingSource informal protocol.

draggingSourceOperationMask

– (unsigned int)**draggingSourceOperationMask**

Returns the dragging operation mask declared by the dragging source (through its **draggingSourceOperationMaskForLocal:** method). If the source permits dragging operations, the elements in the mask will be one or more of the following, combined using the C bitwise OR operator:

Option	Meaning
NSDragOperationCopy	The data represented by the image can be copied.
NSDragOperationLink	The data can be shared.
NSDragOperationGeneric	The operation can be defined by the destination.
NSDragOperationPrivate	The operation is negotiated privately between the source and the destination.
NSDragOperationAll	Combines all the above.

If the source does not permit any dragging operations, then method should return NSDragOperationNone.

If the user is holding down a modifier key during the dragging session and the source doesn't prohibit modifier keys from affecting the drag operation (through its **ignoreModifierKeysWhileDragging** method), then the operating system combines the dragging operation value that corresponds to the modifier key (see the descriptions below) with the source's mask using the C bitwise AND operator.

On Mach the modifier keys are associated with the dragging operation options shown below,

Modifier Key	Dragging Option
Control	NSDragOperationLink
Alternate	NSDragOperationCopy
Command	NSDragOperationGeneric

while on Windows the modifier keys are associated with the following dragging operation options.

Modifier Key	Dragging Option
Control	NSDragOperationCopy
Shift-Control	NSDragOperationLink
Alternate	NSDragOperationCopy

slideDraggedImageTo:

– (void)**slideDraggedImageTo:**(NSPoint)*aPoint*

Slides the image to *aPoint*, a specified location in the screen coordinate system. This method can be used to snap the image down to a particular location. It should only be invoked from within the destination's implementation of **prepareForDragOperation:**—in other words, after the user has released the image but before it's removed from the screen.

NSDraggingSource

(informal protocol)

Category Of: NSObject

Declared In: AppKit/NSDragging.h

Description

The NSDraggingSource informal protocol declares methods that are implemented by the source object in a dragging session (see the NSDraggingDestination protocol for definitions of dragging terms). The *dragging source* is specified as an argument to the **dragImage:at:offset:event:pasteboard:source: slideBack:** message, sent to a window or view object to initiate the dragging session.

Of the methods declared below, only **draggingSourceOperationMaskForLocal:** must be implemented. The other methods are invoked only if the dragging source implements them. All four methods are invoked automatically during a dragging session—you never send an NSDraggingSource message directly to an object.

Method Types

Specifying dragging options

– draggingSourceOperationMaskForLocal:
– ignoreModifierKeysWhileDragging

Responding to dragging sessions

– draggedImage:beganAt:
– draggedImage:endedAt:deposited:

Instance Methods

draggedImage:beganAt:

– (void)**draggedImage:**(NSImage *)*anImage* **beganAt:**(NSPoint)*aPoint*

Invoked when *anImage* is displayed but before it starts following the mouse. *aPoint* is the origin of the image in screen coordinates. This method provides the source object with an opportunity to respond to the initiation of a dragging session. For example, you might choose to have the source give a visual indication to the user that data is being dragged from the source.

See also: **– convertScreenToBase:** (NSWindow), **– convertBaseToScreen:** (NSWindow), **– convertPoint:fromView:** (NSView), **– convertPoint:toView:** (NSView)

draggedImage:endedAt:deposited:

– (void)**draggedImage:**(NSImage *)*anImage*
endedAt:(NSPoint)*aPoint*
deposited:(BOOL)*flag*

Invoked after *anImage* has been released and the dragging destination has been given a chance to operate on the data it represents. *aPoint* is the location of the image's origin in the screen coordinate system when it was released. A YES value for *flag* indicates that the destination accepted the dragged data, while a NO value indicates that it was rejected.

This method provides the source object with an opportunity to respond to either a successful or a failed dragging session. For example, if you are moving data from one location to another, you could use this method to make the source data disappear from its previous location, if the dragging session is successful, or reset itself to its previous state, in the event of a failure.

See also: **– convertScreenToBase:** (NSWindow), **– convertBaseToScreen:** (NSWindow), **– convertPoint:fromView:** (NSView), **– convertPoint:toView:** (NSView)

draggingSourceOperationMaskForLocal:

– (unsigned int)**draggingSourceOperationMaskForLocal:**(BOOL)*flag*

This is the only NSDraggingSource method that must be implemented by the source object. It should return a mask, built by combining the applicable constants listed below using the C bitwise OR operator. You should use this mask to indicate which types of dragging operations the source object will allow to be performed on the dragged image's data. A YES value for *flag* indicates that the candidate destination object (the window or view over which the dragged image is currently poised) is in the same application as the source, while a NO value indicates that the destination object is in a different application.

Option	Meaning
NSDragOperationCopy	The data represented by the image can be copied.
NSDragOperationLink	The data can be shared.
NSDragOperationGeneric	The operation can be defined by the destination.
NSDragOperationPrivate	The operation is negotiated privately between the source and the destination.
NSDragOperationAll	Combines all the above.

If the source does not permit any dragging operations, then it should return NSDragOperationNone.

ignoreModifierKeysWhileDragging

– (BOOL)**ignoreModifierKeysWhileDragging**

Sets whether the use of the modifier keys should have no effect on the type of operation performed. If this method is not implemented or returns NO, then the user can tailor the drag operation by holding down a modifier key during the drag. The dragging option that corresponds to the modifier key is combined with the source's mask (as set with the **draggingSourceOperationMaskForLocal:** method) using the C bitwise AND operator. See the description for the **draggingSourceOperationMask** method in the NSDraggingInfo protocol specification for more information about dragging masks and modifier keys.

NSIgnoreMisspelledWords

Adopted By: NSText

Declared In: AppKit/NSSpellProtocol.h

Protocol Description

Implement this protocol to have the Ignore button in the Spelling panel function properly. The Ignore button allows the user to accept a word that the spelling checker believes is misspelled. In order for this action to update the “ignored words” list for the document being checked, the NSIgnoreMisspelledWords protocol must be implemented.

This protocol is necessary because a list of ignored words is useful only if it pertains to the entire document being checked, but the spelling checker (NSSpellChecker object) does not check the entire document for spelling at once. The spelling checker returns as soon as it finds a misspelled word. Thus, it checks only a subset of the document at any one time. The user usually wants to check the entire document, so usually several spelling checks are run in succession until no misspelled words are found. This protocol allows the list of ignored words to be maintained per-document, even though the spelling checks are not run per-document.

The NSIgnoreMisspelledWords protocol specifies a method, **ignoreSpelling:**, which should be implemented like this:

```
- (void)ignoreSpelling:(id)sender
{
    [[NSSpellChecker sharedSpellChecker] ignoreWord:[[sender selectedCell]
        stringValue] inSpellDocumentWithTag:myDocumentTag];
}
```

The second argument to the NSSpellChecker method **ignoreWord:inSpellDocumentWithTag:** is a tag that the NSSpellChecker can use to distinguish the documents being checked. (See the discussion of “Matching a List of Ignored Words With the Document It Belongs To” in the description of the NSSpellChecker class.) Once the NSSpellChecker has a way to distinguish the various documents, it can append new ignored words to the appropriate list.

To make the ignored words feature useful, the application must store a document’s ignored words list with the document. See the NSSpellChecker class description for more information.

Instance Methods

ignoreSpelling:

– (void)**ignoreSpelling:**(id)*sender*

Implement to allow an application to ignore misspelled words on a document-by-document basis. This message is sent by the NSSpellChecker instance to the object whose text is being checked.

Implement this method by using the code shown in the protocol description.

NSMenuItem

Adopted By: NSMenuItem

Conforms To: NSCoding,
NSCopying
NSObject (NSObject)

Declared In: AppKit/NSMenuItem.h

Warning: The NSMenuItem protocol will be removed from the Application Kit in the Premier release of Rhapsody. The NSMenuItem class will solely assume all associated functionality. This change does not affect binary compatibility between different versions of projects, but might cause failures in project builds. To adapt your projects to this change, alter all references to the protocol (for example, “id <NSMenuItem>”) to references to the class (“NSMenuItem”).

Protocol Description

The NSMenuItem protocol declares methods that are used to manipulate command items in menus. The NSMenuItem class adopts this protocol, implementing all methods the protocol declares, and provides the basic functionality of command items. With some implementations of the OpenStep specification (including OPENSTEP), you cannot replace the NSMenuItem class with a different class which conforms to the NSMenuItem protocol. You may, however, subclass the NSMenuItem class if necessary.

The methods declared by the NSMenuItem protocol allow you to set the titles, actions, targets, tags, images, enabled states, and similar attributes of individual menu items, as well as to obtain the current values of these attributes. As implemented for the NSMenuItem class, a menu item, whenever one of its attributes changes, notifies the associated NSMenu via the **itemChanged:** method. The protocol also allows a conforming object to set keyboard equivalents and (for Microsoft Windows) mnemonics for menu items. See the sections below for more on this functionality.

See the NSMenu, NSMenuView, and NSMenuItemCell class specifications and the NSMenuValidation protocol specification for more information on menus.

Keyboard Equivalents

An object conforming to the NSMenuItem protocol can be assigned a keyboard eqivalent, so that when the user types a character the menu item’s action is sent. The keyboard eqivalent is defined in two parts. First is the basic key equivalent, which must be a Unicode character that can be generated by a single key press without modifier keys (Shift excepted). It is also possible to use a sequence of Unicode characters so long

as the user's key mapping is able to generate the sequence with a single key press. The basic key equivalent is set using **setKeyEquivalent:** and returned by **keyEquivalent**. The second part defines the modifier keys that must also be pressed. This is set using **setKeyEquivalentModifierMask:** and returned by **keyEquivalentModifierMask**. The modifier mask by default includes NSCommandKeyMask, and may also include the masks for the Shift, Alternate, or other modifier keys. Specifying keyboard equivalents in two parts allows you to define a modified keyboard equivalent without having to know which character is generated by the basic key plus the modifier. For example, you can define the keyboard equivalent Command-Alt-f without having to know which character is generated by typing Alt-f.

Certain methods in the NSMenuItem protocol can override assigned keyboard equivalents with those the user has specified in the defaults system. The **setUsesUserKeyEquivalents:** protocol method turns this behavior on or off, and **usesUserKeyEquivalents** returns its status. To determine the user-defined key equivalent for an NSMenuItem object, invoke the **userKeyEquivalent** instance method. If user-defined key equivalents are active and an NSMenuItem object has a user-defined key equivalent, its **keyEquivalent** method returns the user-defined key equivalent and not the one set using **setKeyEquivalent:**.

Mnemonics

On certain platforms, currently including Microsoft Windows, an object conforming to the NSMenuItem protocol can also be assigned a mnemonic. Mnemonics can be assigned on other platforms as well, however, they won't have any effect. Mnemonics are represented by an underlined character in the title of a menu item. The mnemonic can be any character that can be generated by a single key press without modifier keys (Shift excepted). When the menu is active, the user can type the underlined character in the menu item in order to activate that menu item. On Microsoft Windows a user activates the menu by pressing the Alternate key. A particular mnemonic character should only be used once within the set of menu items contained either in the same menu as the menu item or in the application's main menu.

Radio-Style Grouping

By using a few methods of the NSMenuItem protocol, you can implement radio-style groupings of menu commands. In other words, you can have a grouping of menu commands (usually segregated visually with separator items) and only one command in the group can be selected; the selected item is marked by an image, usually a radio-button image, but sometimes a checkmark. If the user selects another command in the group, the previous command is unmarked and the selected command displays the image. As an example of a radio-style grouping, a game could have three commands to indicate the level of play: Beginner, Intermediate, and Advanced.

To implement this feature, first set the images you want to use for the possible command states: "on," "off," and "mixed" (the last is useful for triple-state or indeterminate situations). To set the image, use the commands **setOnStateImage:**, **setOffStateImage:**, and **setMixedStateImage:**. The default image for the "on" state is a checkmark (NSMenuCheckmark) and for the "mixed" state the image is a dash (NSMenuMixedState). The "off" state typically has no image. The radio-button image (which you must set explicitly) is NSMenuRadio.

In an action method that responds to all commands in the group use **setState:** to uncheck the menu item that is currently marked:

```
[curItem setState:NSOffState];
```

Then mark the newly selected command:

```
[sender setState:NSOnState];
```

Method Types

Creating conforming NSMenuItem objects

– initWithTitle:action:keyEquivalent:

Enabling a menu item

– setEnabled:
– isEnabled

Setting the target and action

– setTarget:
– target
– setAction:
– action

Setting the title

– setTitle:
– title

Setting the tag

– setTag:
– tag

Setting the state

– setState:
– state

Setting the image

– setImage:
– image
– setOnStateImage:
– onStateImage
– setOffStateImage:
– offStateImage
– setMixedStateImage:
– mixedStateImage

Managing submenus

– setSubmenu:
– submenu
– hasSubmenu

Getting a separator item

+ separatorItem
– isSeparatorItem

Setting the owning menu

– setMenu:
– menu

Managing key equivalents

– setKeyEquivalent:
– keyEquivalent
– setKeyEquivalentModifierMask:
– keyEquivalentModifierMask

Managing mnemonics

– setMnemonicLocation:
– mnemonicLocation
– setTitleWithMnemonic:
– mnemonic

Managing user key equivalents

+ setUsesUserKeyEquivalents:
+ usesUserKeyEquivalents
– userKeyEquivalent

Representing an object

– setRepresentedObject:
– representedObject

Class Methods

separatorItem

+ (id <NSMenuItem>)**separatorItem**

Returns a menu item that is used to separate logical groups of menu commands. This menu item is disabled. The default separator item is a simple horizontal line.

See also: – **isSeparatorItem**, – **setEnabled:**

setUsesUserKeyEquivalents:

+ (void)**setUsesUserKeyEquivalents:**(BOOL)*flag*

If *flag* is YES, menu items conform to user preferences for key equivalents; otherwise, the key equivalents originally assigned to the menu items are used.

See also: **+ usesUserKeyEquivalents, – userKeyEquivalent**

usesUserKeyEquivalents

+ (BOOL)**usesUserKeyEquivalents**

Returns YES if menu items conform to user preferences for key equivalents; otherwise, returns NO.

See also: **+ setUsesUserKeyEquivalents:, – userKeyEquivalent**

Instance Methods

action

– (SEL)**action**

Returns the receiver's action method.

See also: **– target, – setAction:**

hasSubmenu

– (BOOL)**hasSubmenu**

Returns YES if the receiver has a submenu, NO if it doesn't.

See also: **– setSubmenu:forItem:**(NSMenu)

image

– (NSImage *)**image**

Returns the image displayed by the receiver, or **nil** if it displays no image.

See also: **– setImage:**

initWithTitle:action:keyEquivalent:

– (id)**initWithTitle:**(NSString *)*itemName*
action:(SEL)*anAction*
keyEquivalent:(NSString *)*charCode*

Returns an initialized instance of an object that conforms to the NSMenuItem protocol. The arguments *itemName* and *charCode* must not be **nil** (if there is no title or key equivalent, specify an empty NSString). The *anAction* argument must be a valid selector or NULL For instances of the NSMenItem class, the default initial state is NSStateOff, the default on-state image is a checkmark, and the default mixed-state image is a dash.

isEnabled

– (BOOL)**isEnabled**

Returns YES if the receiver is enabled, NO if not.

See also: – **setEnabled:**

isSeparatorItem

– (BOOL)**isSeparatorItem**

Returns whether the receiver is a separator item (that is, a menu item used to visually segregate related menu items).

See also: + **separatorItem**

keyEquivalent

– (NSString *)**keyEquivalent**

Returns the receiver's unmodified keyboard equivalent, or the empty string if one hasn't been defined. Use **keyEquivalentModifierMask** to determine the modifier mask for the key equivalent.

See also: **– userKeyEquivalent, – mnemonic, – setKeyEquivalent:**

keyEquivalentModifierMask

– (unsigned int)**keyEquivalentModifierMask**

Returns the receiver's keyboard equivalent modifier mask.

See also: **– setKeyEquivalentModifierMask:**

menu

– (NSMenu *)**menu**

Returns the menu to which the receiver belongs, or **nil** if no menu has been set.

See also: – **setMenu:**

mixedStateImage

– (NSImage *)**mixedStateImage**

Returns the image used to depict a “mixed state.” A mixed state is useful for indicating “off” and “on” attribute values in a group of selected objects, such as a selection of text containing bold and plain (non-bolded) worlds.

See also: – **setMixedStateImage:**

mnemonic

– (NSString *)mnemonic

Returns the character in the menu item title that appears underlined for use as a mnemonic. If there is no mnemonic character, returns an empty string.

See also: – **setTitleWithMnemonic:**

mnemonicLocation

– (unsigned int)**mnemonicLocation**

Returns the position of the underlined character in the menu item title used as a mnemonic. The position is the zero based index of that character in the title string. If the receiver has no mnemonic character, returns NSNotFound.

See also: – **setMnemonicLocation:**

offStateImage

– (NSImage *)offStateImage

Returns the image used to depict the receiver’s “off” state, or **nil** if the image has not been set.

See also: – **setOffStateImage:**

onStateImage

– (NSImage *)**onStateImage**

Returns the image used to depict the receiver's "on" state, or **nil** if the image has not been set.

See also: – **setOnStateImage:**

representedObject

– (id)**representedObject**

Returns the object that the receiving menu item represents. For example, you might have a menu list the names of views that are swapped into the same panel. The represented objects would be the appropriate NSView objects. The user would then be able to swtich back and forth between the different views that are displayed by selecting the various menu items.

See also: – **tag**, – **setRepresentedObject:**

setAction:

– (void)**setAction:**(SEL)*aSelector*

Sets the receiver's action method to *aSelector.*

See also: – **setTarget:**, – **action**

setEnabled:

– (void)**setEnabled:**(BOOL)*flag*

Sets whether the receiver is enabled based on *flag*. If a menu item is disabled, it's keyboard equivalent and mnemonic are also disabled. See the NSMenuValidation informal protocol specification for cautions regarding this method.

See also: – **isEnabled**

setImage:

– (void)**setImage:**(NSImage *)*menuImage*

Set's the receiver's image to *menuImage*. If *menuImage* is **nil**, the current image (if any) is removed. This image is not affected by changes in menu-item state.

See also: – **image**

setKeyEquivalent:

– (void)**setKeyEquivalent:**(NSString *)*aString*

Sets the receiver's unmodified key equivalent to *aString*. If you want to remove the key equivalent from a menu item, pass an empty string (@"") for *aString* (never pass **nil**). Use **setKeyEquivalentModifierMask:** to set the appropriate mask for the modifier keys for the key equivalent.

See also: **– setMnemonicLocation:, – keyEquivalent**

setKeyEquivalentModifierMask:

– (void)**setKeyEquivalentModifierMask:**(unsigned int)*mask*

Sets the receiver's keyboard equivalent modifiers (indicating modifiers such as the Shift or Alternate keys) to those in *mask*. *mask* is an integer bit field containing any of these modifier key masks, combined using the C bitwise OR operator:

NSShiftKeyMask
NSAlternateKeyMask
NSCommandKeyMask

On Mach, you should always set NSCommandKeyMask in *mask*; on Microsoft Windows, this is not required.

NSShiftKeyMask is relevant only for function keys; that is, for key events whose modifier flags include NSFunctionKeyMask. For all other key events NSShiftKeyMask is ignored and characters typed while the Shift key is pressed are interpreted as the shifted versions of those characters; for example, Command-Shift-'c' is interpreted as Command-'C'.

See the NSEvent class specification for more information about modifier mask values.

See also: **– keyEquivalentModifierMask**

setMenu:

– (void)**setMenu:**(NSMenu *)*aMenu*

Sets the receiver's menu to *aMenu*. This method is invoked by the owning NSMenu when the receiver is added or removed. You shouldn't have to invoke this method in your own code, although it can be overriden to provide specialized behavior.

See also: – **menu**

setMixedStateImage:

– (void)**setMixedStateImage:**(NSImage *)*itemImage*

Sets the image of the receiver that indicates a”mixed” state, that is, a state neither “on” or “off.” If *itemImage* is **nil**, any current mixed-state image is removed.

See also: – **mixedStateImage**, – **setOffStateImage:**, – **setOnStateImage:**, – **setState:**

setMnemonicLocation:

– (void)**setMnemonicLocation:**(unsigned int)*location*

Sets the character of the menu item title at *location* that is to be underlined. *location* must be between 0 and 254. This character identifies the access key on Windows by which users can access the menu item.

See also: – **mnemonicLocation**

setOffStateImage:

– (void)**setOffStateImage:**(NSImage *)*itemImage*

Sets the image of the receiver that indicates an “off” state. If *itemImage* is **nil**, any current off-state image is removed.

See also: – **offStateImage**, – **setMixedStateImage:**, – **setOnStateImage:**, – **setState:**

setOnStateImage:

– (void)**setOnStateImage:**(NSImage *)*itemImage*

Sets the image of the receiver that indicates an “on” state. If *itemImage* is **nil**, any current off-state image is removed.

See also: – **onStateImage**, – **setMixedStateImage:**, – **setOffStateImage:**, – **setState:**

setRepresentedObject:

– (void)**setRepresentedObject:**(id)*anObject*

Sets the object represented by the receiver to *anObject*. By setting a represented object for a menu item you make an association between the menu item and that object. The represented object functions as a more specific form of tag that allows you to associate any object, not just an **int**, with the items in a menu.

For example, an NSView object might be associated with a menu item—when the user chooses the menu item, the represented object is fetched and displayed in a panel. Several menu items might control the display of multiple views in the same panel.

See also: – **setTag:**, – **representedObject**

setState:

– (void)**setState:**(int)*itemState*

Sets the state of the receiver to *itemState*, which should be one of NSOffState, NSOnState, or NSMixedState. The image associated with the new state is displayed to the left of the menu item.

See also: – **state**, – **setMixedStateImage:**, – **setOffStateImage:**, – **setOnStateImage:**

setSubmenu:

– (void)**setSubmenu:**(NSMenu *)*aSubmenu*

Sets the submenu of the receiver to *aSubmenu*. The default implementation of the NSMenuItem class raises an exception if *aSubmenu* already has a supermenu.

See also: – **submenu**, – **hasSubmenu**

setTag:

– (void)**setTag:**(int)*anInt*

Sets the receiver's tag to *anInt*.

See also: – **setRepresentedObject:**, – **tag**

setTarget:

– (void)**setTarget:**(id)*anObject*

Sets the receiver's target to *anObject*.

See also: **– setAction:, – target**

setTitle:

– (void)**setTitle:**(NSString *)*aString*

Sets the receiver's title to *aString*.

See also: – **title**

setTitleWithMnemonic:

– (void)**setTitleWithMnemonic:**(NSString *)*aString*

Sets the title of a menu item with a character underlined to denote an access key (Windows only). Use an ampersand character to mark the character (the one following the ampersand) to be underlined. For example, the following message causes the 'c' in 'Receive' to be underlined:

```
[aMenuItem setTitleWithMnemonic:NSLocalizedString(@"Re&ceive")];
```

See also: – **mnemonic**, – **setMnemonicLocation:**

state

– (int)**state**

Returns the state of the receiver, which is NSOffState (the default), NSOnState, or NSMixedState.

See also: – **setState:**

submenu

– (NSMenu *)**submenu**

Returns the submenu associated with the receiving menu item, or **nil** if no submenu is associated with it. In the implementation of the NSMenuItem class, if the receiver responds YES to **hasSubmenu**, the submenu is returned.

See also: – **hasSubmenu**, – **setSubmenu**:

tag

– (int)**tag**

Returns the receiver's tag.

See also: – **representedObject**, – **setTag:**

target

– (id)**target**

Returns the receiver's target.

See also: **– action, – setTarget:**

title

– (NSString *)**title**

Returns the receiver's title.

See also: **– setTitle:**

userKeyEquivalent

– (NSString *)**userKeyEquivalent**

Returns the user-assigned key equivalent for the receiver.

See also: **– keyEquivalent**

NSMenuValidation

(informal protocol)

Category Of: NSObject

Declared In: AppKit/NSMenu.h

Protocol Description

This informal protocol allows your application to update the enabled or disabled status of an NSMenuItem. It declares only one method, **validateMenuItem:**. By default, every time a user event occurs, NSMenu automatically enables and disables each visible menu item based on criteria described in "Automatic Updating of NSMenuItems," below. Implement **validateMenuItem:** in cases where you want to override NSMenu's default enabling scheme.

NSMenuItems can be enabled or disabled in two ways: explicitly, by sending the **setEnabled:** message, or automatically, as described below. Automatic updating can be turned on and off with NSMenu's **setAutoenablesItems:** message.

Automatic Updating of NSMenuItems

Whenever a user event occurs, the NSMenu object updates the status of every one of its visible menu items. To update the status of a menu item, an NSMenu tries to find the object that responds to the NSMenuItem's action message. It searches the following objects in the given order until it finds one that responds to the action message (note that it doesn't actually send the action message):

- The NSMenuItem's target. If the target is non-**nil**, the search ends here whether the target responds or not.
- The key window's responder chain, starting with its first responder.
- The key window itself.
- The key window's delegate.
- The main window's responder chain, starting with its first responder.
- The main window itself.
- The main window's delegate.
- The NSApplication object.
- The NSApplication object's delegate.

If none of these objects responds to the action message, the menu item is disabled. If NSMenu finds an object that responds to the action message, it then checks to see if that object responds to the **validateMenuItem:** method (the method declared in this informal protocol). If **validateMenuItem:** is not

implemented in that object, the menu item is enabled. If it is implemented, the return value of **validateMenuItem:** indicates whether the menu item should be enabled or disabled.

Here is an example of using **validateMenuItem:** to override automatic enabling. If your application has a Copy menu item that sends the **copy:** action message to the first responder, that menu item is automatically enabled any time an object that responds to **copy:**, such as an NSText object, is the first responder of the key or main window. If you create a class whose instances might become the first responder, and which doesn't support copying of everything it allows the user to select, you should implement **validateMenuItem:** in that class. **validateMenuItem:** will then return NO if items that can't be copied are selected (or if no items are selected) and YES if all items in the selection can be copied. By implementing **validateMenuItem:**, you can have the Copy menu item disabled even though the target object does implement the **copy:** method. If a class never permits copying, then you simply omit an implemention of **copy:** in that class, and the Copy menu item is disabled automatically whenever an instance of that class is the first responder.

If you send a **setEnabled:** message to enable or disable a menu item when automatic updating is turned on (with NSMenu's **setAutoEnablesItems:**), other objects might undo what you have done after another user event occurs. Hence you can never be sure that the menu item will remain the way you set it. If your application must use **setEnabled:**, turn off the automatic enabling of menu items in order to get predictable results.

Instance Methods

validateMenuItem:

– (BOOL)**validateMenuItem:**(NSMenuItem *)*aMenuItem*

Implemented to override the default action of enabling or disabling *aMenuItem*. The object implementing this method must be the target of *aMenuItem*. It returns YES to enable the *aMenuItem*, NO to disable it. You can determine which menu item *aMenuItem* is by querying it for its title, tag, or action.

The following example beeps and disables the menu item "Next Record" if the selected line in a table view is the last one; conversely, it beeps and disables the menu item "Prior Record" if the selected row is the first one in the table view. (**countryKeys** is an array of names appearing in the table view.)

```
- (BOOL)validateMenuItem:(NSMenuItem *)anItem
{
    int row = [tableView selectedRow];
    if ([[anItem title] isEqualToString:@"Next Record"] &&
        (row == [countryKeys indexOfObject:[countryKeys lastObject]])) {
        return NO;
    }
    if ([[anItem title] isEqualToString:@"Prior Record"] && row == 0 ) {
        return NO;
    }
    return YES;
}
```

NSNibAwaking

(informal protocol)

Category Of: NSObject

Declared In: AppKit/NSNibLoading.h

Protocol Description

This informal protocol consists of a single method, **awakeFromNib**. Classes can implement this method to perform final initialization of state after objects have been loaded from an Interface Builder archive.

Instance Methods

awakeFromNib

– (void)**awakeFromNib**

Implemented to prepare the receiver for service after it has been loaded from an Interface Builder archive, or *nib file*. An **awakeFromNib** message is sent to each object loaded from the archive, but only if it can respond to the message, and only after all the objects in the archive have been loaded and initialized. When an object receives an **awakeFromNib** message, it's guaranteed to have all its outlet instance variables set.

Note: This method is also sent during Interface Builder's test mode to objects instantiated from loaded palettes, which include executable code for the objects. It isn't sent to objects defined solely by using the Classes display of the nib file window in Interface Builder.

When an Interface Builder archive is loaded into an application, each custom object from the archive is first initialized with an **init** message, or **initWithFrame:** if the object is a kind of NSView. It's then more specifically initialized with the properties that it was configured with using Interface Builder. This part of the initialization process uses any **set***Variable***:** methods that are available (where *variable* is the name of an instance variable whose value was set in Interface Builder). Finally, after all the objects are fully initialized, each receives an **awakeFromNib** message.

The order in which objects are loaded from the archive is not guaranteed. Therefore, it's possible for a **set***Variable***:** message to be sent to an object before its companion objects have been unarchived. For this reason, **set***Variable***:** methods should not send messages to other objects in the archive. However, messages to other objects can safely be sent from within **awakeFromNib**—by which time it's assured that all the objects are unarchived and initialized (though not necessarily awakened, of course).

Typically, **awakeFromNib** is implemented for classes whose instances are used as the owners of a loaded nib file (shown as "File's Owner" in Interface Builder). Such a class has the express purpose of connecting the loaded objects with objects in the application, and can thereafter be disposed of, or remain in the

capacity of a controller or coordinator for the loaded objects. For example, suppose that a nib file contains two custom views that must be positioned relative to each other at run time. Trying to position them when either one of the views is initialized (in **initWithCoder:** or a **set***Variable***:** method) might fail, since the other views might not be unarchived and initialized yet. However, it can be done in the nib file owner's **awakeFromNib** method (**firstView** and **secondView** are outlets of the file's owner):

```
- (void)awakeFromNib
{
    NSRect viewFrame;

    if ([[self superclass] instancesRespondToSelector:@selector(awakeFromNib)]) {
        [super awakeFromNib];
    }
    viewFrame = [firstView frame];
    viewFrame.origin.x += viewFrame.size.width;
    [secondView setFrame:viewFrame];
    return;
}
```

Note the testing of the superclass before invoking its implementation of **awakeFromNib**. The Application Kit declares a prototype for this method, but doesn't implement it. Because there's no default implementation of **awakeFromNib**, be sure to invoke it only when the object does in fact respond.

See also: + **loadNibNamed:owner:** (NSBundle Additions),
– **awakeAfterUsingCoder** (NSObject class of the Foundation Kit),
– **initWithCoder:** (NSCoding protocol of the Foundation Kit),
+ **initialize** (NSObject class of the Foundation Kit)

NSServicesRequests

(informal protocol)

Category Of: NSObject

Declared In: AppKit/NSApplication.h

Protocol Description

This informal protocol consists of two methods, **writeSelectionToPasteboard:types:** and **readSelectionFromPasteboard:**. The first is implemented to provide data to a remote service, and the second to receive any data the remote service might send back. Both respond to messages that are generated when the user chooses a command from the Services menu.

Instance Methods

readSelectionFromPasteboard:

- (BOOL)**readSelectionFromPasteboard:**(NSPasteboard *)*pboard*

Implemented to replace the current selection (that is, the text or objects that are currently selected) with data read from the Pasteboard object *pboard*. The data would have been placed in the pasteboard by another application in response to a remote message from the Services menu. A **readSelectionFromPasteboard:** message is sent to the same object that previously received a **writeSelectionToPasteboard:types:** message.

There's no default **readSelectionFromPasteboard:** method. The Application Kit declares a prototype for this method, but doesn't implement it.

writeSelectionToPasteboard:types:

- (BOOL)**writeSelectionToPasteboard:**(NSPasteboard *)*pboard* **types:**(NSArray *)*types*

Implemented to write the current selection to the Pasteboard object *pboard*. The selection should be written as one or more of the data types listed in *types*. After writing the data, this method should return YES. If for any reason it can't write the data, it should return NO.

A **writeSelectionToPasteboard:types:** message is sent to the first responder when the user chooses a command from the Services menu, but only if the receiver didn't return **nil** to a previous **validRequestorForSendType:returnType:** message.

After this method writes the data to the pasteboard, a remote message is sent to the application that provides the service the user requested. If the service provider supplies return data to replace the selection, the first responder will then receive a **readSelectionFromPasteboard:** message.

There's no default **writeSelectionToPasteboard:types:** method. The Application Kit declares a prototype for this method, but doesn't implement it.

See also: – **validRequestorForSendType:returnType:** (NSResponder class)

NSTableDataSource

(informal protocol)

Category Of: NSObject

Declared In: AppKit/NSTableView.h

Category Description

The NSTableDataSource category declares the methods that an NSTableView uses to access the contents of its data source object. It determines how many rows to display by sending a **numberOfRowsInTableView:** message, and accesses individual values with the **tableView: objectValueForTableColumn:row:** and **tableView:setObjectValue:forTableColumn:row:** methods. A data source must implement the first two methods to work with an NSTableView, but if it doesn't implement the third the NSTableView simply provides read-only access to its contents.

The NSTableView treats objects provided by its data source as values to be displayed in NSCell objects. If these objects aren't of common value classes—such as NSString, NSNumber, and so on—you'll need to create a custom NSFormatter to display them. See the NSFormatter class specification for more information.

Suppose that an NSTableView's column identifiers are set up as NSStrings containing the names of attributes for the column, such as "Last Name", "City", and so on, and that the data source stores its records as an NSMutableArray, called **records**, of NSMutableDictionary objects using those names as keys. Here's a small example, given as an ASCII property list:

```
(
    {
        "Last Name" = Anderson;
        "First Name" = James;
        Abode = apartment;
        City = "San Francisco";
    },
    {
        "Last Name" = Beresford;
        "First Name" = Keith;
        Abode = apartment;
        City = "Redwood City";
    }
)
```

With such a record structure, this implementation of **tableView:objectValueForTableColumn:row:** suffices to retrieve values for the NSTableView:

```
- (id)tableView:(NSTableView *)aTableView
    objectValueForTableColumn:(NSTableColumn *)aTableColumn
    row:(int)rowIndex
{
    id theRecord, theValue;

    NSParameterAssert(rowIndex >= 0 && rowIndex < [records count]);
    theRecord = [records objectAtIndex:rowIndex];
    theValue = [theRecord objectForKey:[aTableColumn identifier]];
    return theValue;
}
```

Here's the corresponding method for setting values:

```
- (void)tableView:(NSTableView *)aTableView
    setObjectValue:anObject
    forTableColumn:(NSTableColumn *)aTableColumn
    row:(int)rowIndex
{
    id theRecord;

    NSParameterAssert(rowIndex >= 0 && rowIndex < [records count]);
    theRecord = [records objectAtIndex:rowIndex];
    [theRecord setObject:anObject forKey:[aTableColumn identifier]];
    return;
}
```

Finally, **numberOfRowsInTableView:** simply returns the count of the NSArray:

```
- (int)numberOfRowsInTableView:(NSTableView *)aTableView
{
    return [records count];
}
```

In each case, the NSTableView that sends the message is provided as *aTableView*. A data source object that manages several sets of data can choose the appropriate set based on which NSTableView sends the message.

Method Types

Getting values

– numberOfRowsInTableView:
– tableView:objectValueForTableColumn:row:

Setting values

– tableView:setObjectValue:forTableColumn:row:

Instance Methods

numberOfRowsInTableView:

– (int)**numberOfRowsInTableView:**(NSTableView *)*aTableView*

Returns the number of records managed for *aTableView* by the data source object. An NSTableView uses this method to determine how many rows it should create and display.

tableView:objectValueForTableColumn:row:

– (id)**tableView:**(NSTableView *)*aTableView*
objectValueForTableColumn:(NSTableColumn *)*aTableColumn*
row:(int)*rowIndex*

Returns an attribute value for the record in *aTableView* at *rowIndex*. *aTableColumn* contains the identifier for the attribute, which you get by using NSTableColumn's **identifier** method. For example, if *aTableColumn* stands for the city that an employee lives in and *rowIndex* specifies the record for an employee who lives in Portland, this method returns an object with a string value of "Portland". See the category description for an example.

tableView:setObjectValue:forTableColumn:row:

– (void)**tableView:**(NSTableView *)*aTableView*
setObjectValue:(id)*anObject*
forTableColumn:(NSTableColumn *)*aTableColumn*
row:(int)*rowIndex*

Sets an attribute value for the record in *aTableView* at *rowIndex*. *anObject* is the new value, and *aTableColumn* contains the identifier for the attribute, which you get by using NSTableColumn's **identifier** method. See the category description for an example.

NSTextAttachmentCell

Adopted By: NSTextAttachmentCell

Declared In: AppKit/NSTextAttachment.h

Protocol Description

The NSTextAttachmentCell protocol declares the interface for objects that draw text attachment icons and handle mouse events on their icons. With the exceptions of **cellBaselineOffset:**, **setAttachment:** and **attachment**, all of these methods are implemented by the NSCell class and described in that class specification.

See the NSAttributedString and NSTextView class specifications for general information on text attachments.

Method Types

Drawing

– drawWithFrame:inView:
– highlight:withFrame:inView:

Cell size and position

– cellSize
– cellBaselineOffset

Event handling

– wantsToTrackMouse
– trackMouse:inRect:ofView:untilMouseUp:

Setting the attachment

– setAttachment:
– attachment

Instance Methods

attachment

– (NSTextAttachment *)**attachment**

Returns the text attachment object that owns the receiver.

See also: – **setAttachment:**

cellBaselineOffset

– (NSPoint)**cellBaselineOffset**

Returns the position where the attachment cell's image should be drawn in text, relative to the current point established in the glyph layout. The image should be drawn so that its lower left corner lies on this point.

See also: – **icon** (NSFileWrapper)

cellSize

– (NSSize)**cellSize**

Returns the size of the attachment's icon.

See also: – **icon** (NSFileWrapper), – **fileWrapper** (NSTextAttachment)

drawWithFrame:inView:

– (void)**drawWithFrame:**(NSRect)*cellFrame* **inView:**(NSView *)*aView*

Draws the receiver's image within *cellFrame* in *aView*, which should be the focus view.

See also: – **drawWithFrame:inView:** (NSCell), – **lockFocus** (NSView)

highlight:withFrame:inView:

– (void)**highlight:**(BOOL)*flag*
withFrame:(NSRect)*cellFrame*
inView:(NSView *)*aView*

Draws the receiver's image—with highlighting if *flag* is YES—within *cellFrame* in *aView*, which should be the focus view.

See also: – **highlight:withFrame:inView:** (NSCell), – **lockFocus** (NSView)

setAttachment:

– (void)**setAttachment:**(NSTextAttachment *)*anAttachment*

Sets the text attachment object that owns the receiver to *anAttachment*, without retaining it (the text attachment, as the owner, retains the cell).

See also: – **attachment**, – **setAttachmentCell:** (NSTextAttachment)

trackMouse:inRect:ofView:untilMouseUp:

– (BOOL)**trackMouse:**(NSEvent *)*theEvent*
inRect:(NSRect)*cellFrame*
ofView:(NSView *)*aTextView*
untilMouseUp:(BOOL)*flag*

Handles a mouse-down event on the receiver's image. *theEvent* is the mouse-down event. *cellFrame* is the region of *aTextView* in which further mouse events should be tracked. *aTextView* is the view which received the event. It's assumed to be an NSTextView, and should be the focus view. If *flag* is YES, the receiver tracks the mouse until a mouse-up event occurs; if *flag* is NO, it stops tracking when a mouse-dragged event occurs outside of *cellFrame*. Returns YES if the receiver successfully finished tracking the mouse (typically through a mouse-up event), NO otherwise (such as when the mouse is dragged outside *cellFrame*).

NSTextAttachmentCell's implementation of this method calls upon *aTextView*'s delegate to handle the event. If *theEvent* is a mouse-up event for a double click, the text attachment cell sends the delegate a **textView:doubleClickedOnCell:inRect:** message and returns YES. Otherwise, depending on whether the user clicks or drags the cell, it sends the delegate a **textView:clickedOnCell:inRect:** or a **textView: draggingCell:inRect:event:** message and returns YES. NSTextAttachmentCell's implementation returns NO only if *flag* is NO and the mouse is dragged outside of *cellFrame*. The delegate methods are invoked only if the delegate responds.

See also: – **wantsToTrackMouse**, – **trackMouse:inRect:ofView:untilMouseUp:** (NSCell),
– **lockFocus** (NSView)

wantsToTrackMouse

– (BOOL)**wantsToTrackMouse**

Returns YES if the receiver will handle a mouse event occurring over its image (to support dragging, for example), NO otherwise. NSTextAttachmentCell's implementation of this method returns YES. The NSView containing the cell should invoke this method before sending a **trackMouse:inRect:ofView: untilMouseUp:** message.

For an attachment in an attributed string, if the attachment cell returns NO its attachment character should be selected rather than the cell being asked to track the mouse. this results in the attachment icon behaving as any regular glyph in text.

NSTextInput

Adopted By: NSInputManager

Declared In: AppKit/NSInputManager.h

Protocol Description

Note: *This class specification is incomplete and has not received a technical review. It is included in this release to test the linkage between the application development tools and the on-line documentation. What information it contains should be considered preliminary and subject to change.*

The methods of the NSTextInput protocol are implemented by objects—for example, responders—that handle text input. A client object such as a text view (“the client object”) that speaks this protocol must be reponsible for the following things:

1. The client object must maintain a “marked region” within which text input and, possibly, character conversion take place. The marked region may have a length of zero. The client object must maintain an “insertion point”—typically at the end of the marked region, though it may be within the region. The “selection” within the client object, if any, is entirely contained within the marked region whenever there is a marked region.2. The client object is responsible for sending messages to currentInputManager when the mouse goes down inside the marked region, or when the mouse leaves the marked region. Within the marked region, this allows the selection to be changed. Out of the marked region, it allows the region to be “abandoned”. (See below.)

3. When there is a non-zero marked region, the client object is responsible for notifying the input manager when the selection changes, or when other programmatic changes to the text affect the marked region. It can do this by sending a message to set the selection, abandon the marked region, etc.

4. When the client object relinquishes first responder, it will typically send markedTextWillBeAbandoned: to the currentInputManager. It must send markedTextWillBeAbandoned: when its insertion point (or selection moves outside the marked range. The server will typically respond by simply unmarking the region, but may remove the marked region's text entirely.

Note: If this protocol is not implemented by a client object that does have a keyDown: method, then in-line input is not possible for that client object, and will have to be handled externally.

The NSTextInput protocol is implemented by a IM to receive input from the server on behalf of the current client, and otherwise mediate between the client object and the server. It then forwards the corresponding messages to the client, or gets information from the client to pass back to the server, as appropriate.

The message passing between NSApp, IM, UIobj, and Server is all synchronous. That is, e.g., when IM sends a message to Server, any reply comes back and is relayed to UIobj before the original message returns.

A key binding manager splits the stream of keyDown: messages (intercepted by NSResopnder) into commands and text. If there is an Input Manager in the loop, it will further use any of these NSTextInput messages to control the marked region.

Method Types

Marking text

– setMarkedText:selectedRange:
– getMarkedText:selectedRange:
– hasMarkedText
– unmarkText

Other

– conversationIdentifier
– doCommandBySelector:
– insertText:

Instance Methods

conversationIdentifier

– (long)**conversationIdentifier**

Returns a number used to identify the receiver's input management session to the input server.

<<more information forthcoming>>

doCommandBySelector:

– (void)**doCommandBySelector:**(SEL)*aSelector*

Attempts to invoke *aSelector* or pass the message up the responer chain. This method is invoked by an input manager in response to an **interpretKeyEvents:** message.

<<more information forthcoming>>

See also: – **interpretKeyEvents:** (NSResponder)

getMarkedText:selectedRange:

– (void)**getMarkedText:**(out NSString **)*aString* **selectedRange:**(out NSRange *)*aRange*

Returns by reference in *aString* the receiver's marked text, if any, and in *aRange* the range of the selection within *aString* (not in terms of the receiver's entire text stream).

hasMarkedText

– (BOOL)**hasMarkedText**

Returns YES if the receiver has text that's still being interpreted by the input manager, NO if it doesn't.

<<more information forthcoming>>

insertText:

– (void)**insertText:**(NSString *)*aString*

Inserts *aString* into the receiver's text stream. This method is invoked by an input manager in response to an **interpretKeyEvents:** message.

<<more information forthcoming>>

See also: – **interpretKeyEvents:** (NSResponder)

setMarkedText:selectedRange:

– (void)**setMarkedText:**(NSString *)*aString* **selectedRange:**(NSRange)*selRange*

<<forthcoming>>

unmarkText

– (void)**unmarkText**

Removes any marking from pending input text, and accepts the text in its current state.

<<more information forthcoming>>

Defined Types

NSBorderType

DECLARED IN AppKit/NSView.h

SYNOPSIS

```
typedef enum _NSBorderType {
    NSNoBorder,
    NSLineBorder,
    NSBezelBorder,
    NSGrooveBorder
} NSBorderType;
```

DESCRIPTION This type represents the kinds of border that can be drawn around certain NSView subclasses.

NSButtonType

DECLARED IN AppKit/NSButtonCell.h

SYNOPSIS

```
typedef enum _NSButtonType {
    NSMomentaryPushButton,
    NSPushOnPushOffButton,
    NSToggleButton,
    NSSwitchButton,
    NSRadioButton,
    NSMomentaryChangeButton,
    NSOnOffButton,
    NSMomentaryLight,
} NSButtonType;
```

DESCRIPTION This type represents the way NSButtons and NSButtonCells behave when pressed, and the way they display their state. See NSButton's and NSButtonCell's **setButtonType:** methods for more information.

NSCellAttribute

DECLARED IN AppKit/NSCell.h

SYNOPSIS

```
typedef enum _NSCellAttribute {
    NSCellDisabled,
    NSCellState,
    NSPushInCell,
    NSCellEditable,
    NSChangeGrayCell,
    NSCellHighlighted,
    NSCellLightsByContents,
    NSCellLightsByGray,
    NSChangeBackgroundCell,
    NSCellLightsByBackground,
    NSCellIsBordered,
    NSCellHasOverlappingImage,
    NSCellHasImageHorizontal,
    NSCellHasImageOnLeftOrBottom,
    NSCellChangesContents,
    NSCellIsInsetButton,
    NSCellAllowsMixedState
} NSCellAttribute;
```

DESCRIPTION This is the type of the first argument to the NSCell methods **setCellAttribute:to:** and **cellAttribute:** methods. Some of the values apply not to NSCell but to one of its subclasses.

Often it's preferable to change cell attributes using more specialized methods like **setState:** or **setEditable:**.

NSCellImagePosition

DECLARED IN AppKit/NSCell.h

SYNOPSIS

typedef enum _NSCellImagePosition {
    NSNoImage,
    NSImageOnly,
    NSImageLeft,
    NSImageRight,
    NSImageBelow,
    NSImageAbove,
    NSImageOverlaps,
} **NSCellImagePosition**;

DESCRIPTION These constants represent the position of an NSButtonCell's NSImage relative to its title. See NSButton's and NSButtonCell's **setImagePosition:** and **imagePosition** methods for more information.

NSCellType

DECLARED IN AppKit/NSCell.h

SYNOPSIS

typedef enum _NSCellType {
    NSNullCellType,
    NSTextCellType,
    NSImageCellType,
} **NSCellType**;

DESCRIPTION The values of this type determine what kind of data an NSCell displays. NSCells of type NSTextCellType and NSImageCellType display text and images, respectively. NSCells of type NSNullCellType display nothing. See the NSCell methods **type** and **setType:** for more information.

NSDataLinkDisposition

DECLARED IN AppKit/NSDataLink.h

SYNOPSIS

```
typedef enum _NSDataLinkDisposition {
    NSLinkInDestination,
    NSLinkInSource,
    NSLinkBroken
} NSDataLinkDisposition;
```

DESCRIPTION Returned by NSDataLink's **disposition** method to identify a link as a destination link, a source link, or a broken link. See the NSDataLink class specification for more information on the dispositions of links.

NSDataLinkNumber

DECLARED IN AppKit/NSDataLink.h

SYNOPSIS

```
typedef int NSDataLinkNumber;
```

DESCRIPTION The type returned by NSDataLink's **linkNumber** method as a persistent identifier of a destination link.

NSDataLinkUpdateMode

DECLARED IN AppKit/NSDataLink.h

SYNOPSIS

```
typedef enum _NSDataLinkUpdateMode {
    NSUpdateContinuously,
    NSUpdateWhenSourceSaved,
    NSUpdateManually,
    NSUpdateNever
} NSDataLinkUpdateMode;
```

DESCRIPTION Used by NSDataLink's **setUpdateMode:** and **updateMode** methods to identify when a link's data is to be updated.

NSEventType

DECLARED IN AppKit/NSEvent.h

SYNOPSIS

```
typedef enum _NSEventType {
    NSLeftMouseDown,
    NSLeftMouseUp,
    NSRightMouseDown,
    NSRightMouseUp,
    NSMouseMoved,
    NSLeftMouseDragged,
    NSRightMouseDragged,
    NSMouseEntered,
    NSMouseExited,
    NSKeyDown,
    NSKeyUp,
    NSFlagsChanged,
    NSAppKitDefined,
    NSSystemDefined,
    NSApplicationDefined
    NSPeriodic,
    NSCursorUpdate
} NSEventType;
```

DESCRIPTION This type represents various kinds of events. It is the return type of NSEvent's type method, and the type of the first argument to NSEvent's **...EventWithType:** methods.

NSFontAction

DECLARED IN AppKit/NSFontManager.h

SYNOPSIS

typedef enum _NSFontAction {
    NSNoFontChangeAction,
    NSViaPanelFontAction,
    NSAddTraitFontAction,
    NSSizeUpFontAction,
    NSSizeDownFontAction,
    NSHeavierFontAction,
    NSLighterFontAction,
    NSRemoveTraitFontAction
} **NSFontAction**;

DESCRIPTION Values of this type tag the actions of font menu cells. When a font menu cell sends a message to NSFontManager, NSFontManager checks the cell for one of these tags.

This type is in the API for explanatory purposes only. You will never use it directly.

NSFontTraitMask

DECLARED IN AppKit/NSFontManager.h

SYNOPSIS typedef unsigned int **NSFontTraitMask;**

DESCRIPTION An NSFontTraitMask characterizes one or more of a font's traits. It's used as an argument type for NSAttributedString's **applyFontTraits:range:** method, NSCStringText's **setSelFontStyle:** method, and several of the methods in the NSFontManager class.

NSGlyph

DECLARED IN AppKit/NSFont.h

SYNOPSIS typedef unsigned int **NSGlyph**;

DESCRIPTION This type is used to specify PostScript glyphs in such NSFont methods as **glyphWithName:**.

NSGlyphInscription

DECLARED IN AppKit/NSLayoutManager.h

SYNOPSIS

```
typedef enum {
    NSGlyphInscribeBase,
    NSGlyphInscribeBelow,
    NSGlyphInscribeAbove,
    NSGlyphInscribeOverstrike,
    NSGlyphInscribeOverBelow
} NSGlyphInscription;
```

DESCRIPTION The inscribe attribute of an glyph determines how it is laid out relative to the previous glyph.

NSGlyphRelation

DECLARED IN AppKit/NSFont.h

SYNOPSIS

```
typedef enum _NSGlyphRelation {
    NSGlyphBelow,
    NSGlyphAbove,
} NSGlyphRelation;
```

DESCRIPTION This type specifies the position of a glyph in relation to the base glyph. Parameters of this type are used in the second slot of the NSFont method **positionOfGlyph:withRelation: toBaseGlyph:...**.

NSGradientType

DECLARED IN AppKit/NSButtonCell.h

SYNOPSIS

typedef enum _NSGradientType {
    NSGradientNone,
    NSGradientConcaveWeak,
    NSGradientConcaveStrong,
    NSGradientConvexWeak,
    NSGradientConvexStrong
} **NSGradientType**;

DESCRIPTION This type represents the darkness gradient of an NSButtonCell. A concave gradient is darkest in the top left corner, a convex gradient is darkest in the bottom right corner. A weak gradient has only weak darkness contrast between opposite corners; a strong gradient has strong contrast. See the NSButtonCell methods **gradient** and **setGradient:** for more information.

NSImageAlignment

DECLARED IN AppKit/NSImageCell.h

SYNOPSIS

typedef enum {
    NSImageAlignCenter,
    NSImageAlignTop,
    NSImageAlignTopLeft,
    NSImageAlignTopRight,
    NSImageAlignLeft,
    NSImageAlignBottom,
    NSImageAlignBottomLeft,
    NSImageAlignBottomRight,
    NSImageAlignRight
} **NSImageAlignment**;

DESCRIPTION This type defines the ways of aligning an NSImage within an NSImageCell. It is the return type for NSImageCell's and NSImageView's **imageAlignment** methods, and an argument type for their **setImageAlignment:** methods.

NSImageFrameStyle

DECLARED IN AppKit/NSImageCell.h

SYNOPSIS

```
typedef enum {
    NSImageFrameNone,
    NSImageFramePhoto,
    NSImageFrameGrayBezel,
    NSImageFrameGroove,
    NSImageFrameButton
} NSImageFrameStyle;
```

DESCRIPTION This type defines the kinds of frames that can appear around an NSImageCell. It is the return type for NSImageCell's and NSImageView's **imageFrameStyle** methods, and an argument type for their **setImageFrameStyle:** methods.

NSImageScaling

DECLARED IN AppKit/NSImageCell.h

SYNOPSIS

```
typedef enum {
    NSScaleProportionally,
    NSScaleToFit,
    NSScaleNone
} NSImageScaling;
```

DESCRIPTION This type defines the ways that an image can be scaled to fit an NSImageCell. The value NSScaleProportionally means that the image should be scaled in a way that preserves its proportions. The value NSScaleToFit means that the image should fit the NSView, even if that means its proportions must be distorted. The value NSScaleNone means that the image's size should be preserved, even if it must be clipped to fit the NSView.

NSInterfaceStyle

DECLARED IN AppKit/NSInterfaceStyle.h

SYNOPSIS

typedef enum {
    NSNoInterfaceStyle,
    NSNextStepInterfaceStyle,
    NSWindows95InterfaceStyle,
    NSMacintoshInterfaceStyle
} **NSInterfaceStyle**;

DECLARED IN This type defines the style of an application's user interface. It is returned by the **interfaceStyle** method and taken as an argument by the **setInterfaceStyle:** method. Both of the these methods are in the NSInterfaceStyle category of NSResponder.

For more information, see the function **NSInterfaceStyleForKey**.

NSLineBreakMode

DECLARED IN AppKit/NSParagraphStyle.h

SYNOPSIS

typedef enum _NSLineBreakMode {
    NSLineBreakByWordWrapping,
    NSLineBreakByCharWrapping,
    NSLineBreakByClipping,
    NSLineBreakByTruncatingHead,
    NSLineBreakByTruncatingTail,
    NSLineBreakByTruncatingMiddle
} **NSLineBreakMode**;

DESCRIPTION This type defines the ways that a long paragraph can be broken into lines. The possible values are described below.

Value	Meaning
NSLineBreakByWordWrapping	The default value. At the last possible word boundary, the paragraph wraps to the next line.

Value	Meaning
NSLineBreakByCharWrapping	At the last possible character, the paragraph wraps to the next line.
NSLineBreakByClipping	As much of the paragraph appears as will fit on a single line. This value has the same effect as NSLineBreakByTruncatingTail
NSLineBreakByTruncatingHead	As much of the paragraph appears as will fit on a single line. Characters from the start of the paragraph do not appear.
NSLineBreakByTruncatingTail	As much of the paragraph appears as will fit on a single line. Characters from the end of the paragraph do not appear.
NSLineBreakByTruncatingMiddle	As much of the paragraph appears as will fit on a single line. Characters from the middle of the paragraph do not appear.

NSLineMovementDirection

DECLARED IN AppKit/NSTextContainer.h

SYNOPSIS

```
typedef enum {
    NSLineDoesntMove,
    NSLineMovesLeft,
    NSLineMovesRight,
    NSLineMovesDown,
    NSLineMovesUp
} NSLineMovementDirection;
```

DESCRIPTION This is an argument type for the NSTextContainer method **lineFragmentRectForProposedRect:sweepDirection:movementDirection:remainingRect:**.

NSLineSweepDirection

DECLARED IN AppKit/NSTextContainer.h

SYNOPSIS

typedef enum {
    NSLineSweepLeft,
    NSLineSweepRight,
    NSLineSweepDown,
    NSLineSweepUp
} **NSLineSweepDirection**;

DESCRIPTION This is an argument type for the NSTextContainer method **lineFragmentRectForProposedRect:sweepDirection:movementDirection:remainingRect:**.

NSMatrixMode

DECLARED IN AppKit/NSMatrix.h

SYNOPSIS

typedef enum _NSMatrixMode {
    NSRadioModeMatrix,
    NSHighlightModeMatrix,
    NSListModeMatrix,
    NSTrackModeMatrix
} **NSMatrixMode**;

DESCRIPTION These constants represent the modes of operation of an NSMatrix, as described in the NSMatrix class specification.

NSModalSession

DECLARED IN AppKit/NSApplication.h

SYNOPSIS

typedef struct _NSModalSession ***NSModalSession**;

DESCRIPTION Variables of type NSModalSession point to information used by the system between **beginModalSession:for:** and **endModalSession:** messages.

NSPrinterTableStatus

DECLARED IN AppKit/NSPrinter.h

SYNOPSIS

typedef enum _NSPrinterTableStatus {
    NSPrinterTableOK,
    NSPrinterTableNotFound,
    NSPrinterTableError
} **NSPrinterTableStatus**;

DESCRIPTION These constants are used to describe the state of a printer-information table stored by an NSPrinter object. See the NSPrinter method **statusForTable:** for more information.

NSPrintingOrientation

DECLARED IN AppKit/NSPrintInfo.h

SYNOPSIS

typedef enum _NSPrintingOrientation {
    NSPortraitOrientation,
    NSLandscapeOrientation
} **NSPrintingOrientation**;

DESCRIPTION These constants represent the way a page is oriented for printing. In NSPortraitOrientation, the page is taller than it is wide; in NSLandscapeOrientation, the page is wider than it is tall. See the NSPrintInfo methods **orientation** and **setOrientation:** for more information.

NSPrintingPageOrder

DECLARED IN AppKit/NSPrintOperation.h

SYNOPSIS

typedef enum _NSPrintingPageOrder {
    NSDescendingPageOrder,
    NSSpecialPageOrder,
    NSAscendingPageOrder,
    NSUnknownPageOrder
} **NSPrintingPageOrder**;

DESCRIPTION This type represents the order in which pages are to be printed. The value NSSpecialPageOrder tells the spooler to not rearrange the pages. The value NSUnknownPageOrder means that no page order is written out. See the NSPrintOperation methods **pageOrder** and **setPageOrder:** for more information.

NSPrintingPaginationMode

DECLARED IN AppKit/NSPrintInfo.h

SYNOPSIS

typedef enum _NSPrintingPaginationMode {
    NSAutoPagination,
    NSFitPagination,
    NSClipPagination
} **NSPrintingPaginationMode**;

DESCRIPTION These constants represent the different ways in which an image is divided into pages. The value NSFitPagination forces the image to fit on one page. The value NSClipPagination allows the image to be clipped by the page.See the NSPrintInfo class specification for a fuller explanation.

NSRulerOrientation

DECLARED IN AppKit/NSRulerView.h

SYNOPSIS

```
typedef enum {
    NSHorizontalRuler,
    NSVerticalRuler
} NSRulerOrientation;
```

DESCRIPTION This type defines whether an NSRulerView will be displayed horizontally or vertically. It is the return type of NSRulerView's **orientation** method, and an argument to the NSRulerView methods **setOrientation:** and **initWithScrollView:orientation:**.

NSScrollArrowPosition

DECLARED IN AppKit/NSScroller.h

SYNOPSIS

```
typedef enum _NSScrollArrowPosition {
    NSScrollerArrowsMaxEnd,
    NSScrollerArrowsMinEnd,
    NSScrollerArrowsNone
} NSScrollArrowPosition;
```

DESCRIPTION These constants are used in NSScroller's **setArrowsPosition:** method to set the position of the arrows within the scroller.

NSScrollerArrow

DECLARED IN AppKit/NSScroller.h

SYNOPSIS

```
typedef enum _NSScrollerArrow {
    NSScrollerIncrementArrow,
    NSScrollerDecrementArrow
} NSScrollerArrow;
```

DESCRIPTION This is the type of the first argument to the NSScroller method **drawArrow:highlight:**. The value determines which scroll button is drawn.

NSScrollerPart

DECLARED IN AppKit/NSScroller.h

SYNOPSIS

```
typedef enum _NSScrollerPart {
    NSScrollerNoPart,
    NSScrollerDecrementPage,
    NSScrollerKnob,
    NSScrollerIncrementPage,
    NSScrollerDecrementLine,
    NSScrollerIncrementLine,
    NSScrollerKnobSlot
} NSScrollerPart;
```

DESCRIPTION These constants are used in Scroller's **hitPart** method to identify the part of the Scroller specified in a mouse event.

NSSelectionAffinity

DECLARED IN AppKit/NSTextView.h

SYNOPSIS

```
typedef enum _NSSelectionAffinity {
    NSSelectionAffinityUpstream,
    NSSelectionAffinityDownstream
} NSSelectionAffinity;
```

DESCRIPTION This is the return type of the NSTextView method **selectionAffinity**, and the type of the second argument to the NSTextView method **setSelectedRange:affinity:stillSelecting:**.

NSSelectionDirection

DECLARED IN AppKit/NSWindow.h

SYNOPSIS

typedef enum _NSSelectionDirection {
    NSDirectSelection,
    NSSelectingNext,
    NSSelectingPrevious
} **NSSelectionDirection**;

DESCRIPTION This is the return type of the NSWindow method **keyViewSelectionDirection**.

NSSelectionGranularity

DECLARED IN AppKit/NSTextView.h

SYNOPSIS

typedef enum _NSSelectionGranularity {
    NSSelectByCharacter,
    NSSelectByWord,
    NSSelectByParagraph
} **NSSelectionGranularity**;

DESCRIPTION This is the return type of the NSTextView method selectionGranularity, and the type of arguments to two other NSTextView methods, **setSelectionGranularity:** and **selectionRangeForProposedRange:granularity:**.

NSTextAlignment

DECLARED IN AppKit/NSText.h

SYNOPSIS

typedef enum _NSTextAlignment {
    NSLeftTextAlignment,
    NSRightTextAlignment,
    NSCenterTextAlignment,
    NSJustifiedTextAlignment,
    NSNaturalTextAlignment
} **NSTextAlignment**;

DESCRIPTION Variables of this type are used as arguments and return values for methods that specify text alignment.

NSTextTabType

DECLARED IN AppKit/NSParagraphStyle.h

SYNOPSIS

typedef enum _NSTextTabType {
    NSLeftTabStopType,
    NSRightTabStopType,
    NSCenterTabStopType,
    NSDecimalTabStopType
} **NSTextTabType**;

DESCRIPTION This is the return type of NSTextTab's **tabStopType** method, and an argument to NSTextTab's **initWithType:location:** method.

NSTIFFCompression

SYNOPSIS AppKit/NSBitmapImageRep.h

SYNOPSIS

typedef enum _NSTIFFCompression {
NSTIFFCompressionNone,
NSTIFFCompressionCCITTFAX3,
NSTIFFCompressionCCITTFAX4,
NSTIFFCompressionLZW,
NSTIFFCompressionJPEG,
NSTIFFCompressionNEXT,
NSTIFFCompressionPackBits,
NSTIFFCompressionOldJPEG
} **NSTIFFCompression**;

DESCRIPTION

These constants represent the various TIFF (*tag image file format*) data compression schemes. See the NSBitmapImageRep class specification for their meanings.

NSTitlePosition

DECLARED IN

AppKit/NSBox.h

SYNOPSIS

typedef enum _NSTitlePosition {
NSNoTitle,
NSAboveTop,
NSAtTop,
NSBelowTop,
NSAboveBottom,
NSAtBottom,
NSBelowBottom
} **NSTitlePosition**;

DESCRIPTION

This type represents the locations where an NSBox's title can be placed with respect to its border. Thus, for example, NSAboveTop means the title is above the top of the border, NSAtTop means the title breaks the top border, and so on. See the NSBox methods **titlePosition** and **setTitlePosition:**.

NSTrackingRectTag

DECLARED IN AppKit/NSView.h

SYNOPSIS typedef int **NSTrackingRectTag**;

DESCRIPTION This type describes the rectangle used to track the mouse. See the NSView methods **addTrackingRect:...** and **removeTrackingRect:**.

NSUsableScrollerParts

DECLARED IN AppKit/NSScroller.h

SYNOPSIS

```
typedef enum _NSUsableScrollerParts {
    NSNoScrollerParts,
    NSOnlyScrollerArrows,
    NSAllScrollerParts
} NSUsableScrollerParts;
```

DESCRIPTION This type defines the usable parts of an NSScroller; see the class specification for more information.

NSWindowDepth

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS typedef int **NSWindowDepth**;

DESCRIPTION This type represents the depth, or amount of memory, devoted to a single pixel in a window or screen.

Enumerations

NSApplication—Modal Session Return Values

DECLARED IN AppKit/NSApplication.h

SYNOPSIS

```
enum {
    NSRunStoppedResponse,
    NSRunAbortedResponse,
    NSRunContinuesResponse
};
```

DESCRIPTION Return values for **the NSApplication methods runModalFor:** and **runModalSession:**.

NSAttributedString—Underlining

DECLARED IN AppKit/NSAttributedString.h

SYNOPSIS

```
enum {
    NSSingleUnderlineStyle
};
```

DESCRIPTION This defines the only currently supported value for NSUnderlineStyleAttributeName.

NSButtonCell—State Masks

DECLARED IN AppKit/NSCell.h

SYNOPSIS

```
enum {
    NSNoCellMask,
    NSContentsCellMask,
    NSPushInCellMask,
    NSChangeGrayCellMask,
    NSChangeBackgroundCellMask
};
```

DESCRIPTION These masks are passed to the NSButtonCell methods **highlightsBy:** and **showsStateBy:**.

NSCell—Action Flags

DECLARED IN AppKit/NSEvent.h

SYNOPSIS

```
enum {
    NSLeftMouseDownMask,
    NSLeftMouseUpMask,
    NSRightMouseDownMask,
    NSRightMouseUpMask,
    NSMouseMovedMask,
    NSLeftMouseDraggedMask,
    NSRightMouseDraggedMask,
    NSMouseEnteredMask,
    NSMouseExitedMask,
    NSKeyDownMask,
    NSKeyUpMask,
    NSFlagsChangedMask,
    NSAppKitDefinedMask,
    NSSystemDefinedMask,
    NSApplicationDefinedMask,
    NSPeriodicMask,
    NSCursorUpdateMask,
    NSAnyEventMask
};
```

DESCRIPTION These constants are masks for different kinds of events. You pass them to NSCell's **sendActionOn:** method to indicate when an NSCell should send its action message.

NSCell—Data Entry Types

DECLARED IN AppKit/NSCell.h

SYNOPSIS

```
enum {
    NSAnyType,
    NSIntType,
    NSPositiveIntType,
    NSFloatType,
    NSPositiveFloatType,
    NSDoubleType,
    NSPositiveDoubleType
};
```

DESCRIPTION These constants represent the numeric data types that a text NSCell can accept. See NSCell's **setEntryType:** method for more information.

NSCell—States

DECLARED IN AppKit/NSCell.h

SYNOPSIS

```
enum {
    NSStateMixed,
    NSStateOff,
    NSStateOn
};
```

DESCRIPTION These constants are suggested parameter values for the NSCell method **setState:**.

NSColorPanel—Modes

DECLARED IN AppKit/NSColorPanel.h

SYNOPSIS

```
enum {
    NSGrayModeColorPanel,
    NSRGBModeColorPanel,
    NSCMYKModeColorPanel,
    NSHSBModeColorPanel,
    NSCustomPaletteModeColorPanel,
    NSColorListModeColorPanel,
    NSWheelModeColorPanel
};
```

DESCRIPTION These constants represent the possible modes of an NSColorPanel.

NSColorPanel—Mode Masks

DECLARED IN AppKit/NSColorPanel.h

SYNOPSIS

```
enum {
    NSColorPanelGrayModeMask,
    NSColorPanelRGBModeMask,
    NSColorPanelCMYKModeMask,
    NSColorPanelHSBModeMask,
    NSColorPanelCustomPaletteModeMask,
    NSColorPanelColorListModeMask,
    NSColorPanelWheelModeMask,
    NSColorPanelAllModesMask
};
```

DESCRIPTION These constants provide masks for the NSColorPanel modes.

NSDragging—Operations

DECLARED IN AppKit/NSDragging.h

SYNOPSIS

```
enum {
    NSDragOperationNone,
    NSDragOperationCopy,
    NSDragOperationLink,
    NSDragOperationGeneric,
    NSDragOperationPrivate,
    NSDragOperationAll
};
```

DESCRIPTION These constants define the operations that result from a user's drag. For full descriptions of their meanings and uses, see the method descriptions for **draggingSourceOperationMaskForLocal:** (in the NSDraggingSource protocol), **draggingSourceOperationMask** (in the NSDraggingInfo protocol), or **draggingEntered:** (in the NSDraggingDestination protocol).

NSEvent—Function-Key Unicodes

DECLARED IN AppKit/NSEvent.h

SYNOPSIS

```
enum {
    NSUpArrowFunctionKey = 0xF700,
    NSDownArrowFunctionKey = 0xF701,
    NSLeftArrowFunctionKey = 0xF702,
    NSRightArrowFunctionKey = 0xF703,
    NSF1FunctionKey = 0xF704,
    NSF2FunctionKey = 0xF705,
    NSF3FunctionKey = 0xF706,
    NSF4FunctionKey = 0xF707,
    NSF5FunctionKey = 0xF708,
    NSF6FunctionKey = 0xF709,
    NSF7FunctionKey = 0xF70A,
    NSF8FunctionKey = 0xF70B,
    NSF9FunctionKey = 0xF70C,
    NSF10FunctionKey = 0xF70D,
```

NSF11FunctionKey = 0xF70E,
NSF12FunctionKey = 0xF70F,
NSF13FunctionKey = 0xF710,
NSF14FunctionKey = 0xF711,
NSF15FunctionKey = 0xF712,
NSF16FunctionKey = 0xF713,
NSF17FunctionKey = 0xF714,
NSF18FunctionKey = 0xF715,
NSF19FunctionKey = 0xF716,
NSF20FunctionKey = 0xF717,
NSF21FunctionKey = 0xF718,
NSF22FunctionKey = 0xF719,
NSF23FunctionKey = 0xF71A,
NSF24FunctionKey = 0xF71B,
NSF25FunctionKey = 0xF71C,
NSF26FunctionKey = 0xF71D,
NSF27FunctionKey = 0xF71E,
NSF28FunctionKey = 0xF71F,
NSF29FunctionKey = 0xF720,
NSF30FunctionKey = 0xF721,
NSF31FunctionKey = 0xF722,
NSF32FunctionKey = 0xF723,
NSF33FunctionKey = 0xF724,
NSF34FunctionKey = 0xF725,
NSF35FunctionKey = 0xF726,
NSInsertFunctionKey = 0xF727,
NSDeleteFunctionKey = 0xF728,
NSHomeFunctionKey = 0xF729,
NSBeginFunctionKey = 0xF72A,
NSEndFunctionKey = 0xF72B,
NSPageUpFunctionKey = 0xF72C,
NSPageDownFunctionKey = 0xF72D,
NSPrintScreenFunctionKey = 0xF72E,
NSScrollLockFunctionKey = 0xF72F,
NSPauseFunctionKey = 0xF730,
NSSysReqFunctionKey = 0xF731,
NSBreakFunctionKey = 0xF732,
NSResetFunctionKey = 0xF733,
NSStopFunctionKey = 0xF734,
NSMenuFunctionKey = 0xF735,
NSUserFunctionKey = 0xF736,
NSSystemFunctionKey = 0xF737,
NSPrintFunctionKey = 0xF738,

NSClearLineFunctionKey = 0xF739,
NSClearDisplayFunctionKey = 0xF73A,
NSInsertLineFunctionKey = 0xF73B,
NSDeleteLineFunctionKey = 0xF73C,
NSInsertCharFunctionKey = 0xF73D,
NSDeleteCharFunctionKey = 0xF73E,
NSPrevFunctionKey = 0xF73F,
NSNextFunctionKey = 0xF740,
NSSelectFunctionKey = 0xF741,
NSExecuteFunctionKey = 0xF742,
NSUndoFunctionKey = 0xF743,
NSRedoFunctionKey = 0xF744,
NSFindFunctionKey = 0xF745,
NSHelpFunctionKey = 0xF746,
NSModeSwitchFunctionKey = 0xF747
};

DESCRIPTION These Unicodes (0xF700-0xF8FF) are reserved for function keys on the keyboard. Combined in NSStrings, they may be used in the return value of the NSEvent methods **characters...**, and in parameters of the NSEvent method **keyEventWithType:...characters:...**.

NSEvent—Modifier Flags

DECLARED IN AppKit/NSEvent.h

SYNOPSIS

enum {
NSAlphaShiftKeyMask,
NSShiftKeyMask,
NSControlKeyMask,
NSAlternateKeyMask,
NSCommandKeyMask,
NSNumericPadKeyMask,
NSHelpKeyMask,
NSFunctionKeyMask
};

DESCRIPTION These are device-independent bits found in event modifier flags.

NSEvent—Types Defined by the Application Kit

DECLARED IN AppKit/NSEvent.h

SYNOPSIS

```
enum {
    NSWindowExposedEventType,
    NSApplicationActivatedEventType,
    NSApplicationDeactivatedEventType,
    NSWindowMovedEventType,
    NSScreenChangedEventType
};
```

DESCRIPTION These constants represent the types of events defined by the Application Kit.

Constant	Meaning
NSWindowExposedEventType	A nonretained NSWindow has been exposed
NSApplicationActivatedEventType	The application has been activated
NSApplicationDeactivatedEventType	The application has been deactivated
NSWindowMovedEventType	An NSWindow has moved
NSScreenChangedEventType	An NSWindow has changed screens

NSEvent—Types Defined by the System

DECLARED IN AppKit/NSEvent.h

SYNOPSIS

```
enum {
    NSPowerOffEventType
};
```

DESCRIPTION This constant means that the user is turning off the computer.

NSFont—Traits

DECLARED IN AppKit/NSFontManager.h

SYNOPSIS

```
enum {
    NSItalicFontMask,
    NSBoldFontMask,
    NSUnboldFontMask,
    NSNonStandardCharacterSetFontMask,
    NSNarrowFontMask,
    NSExpandedFontMask,
    NSCondensedFontMask,
    NSSmallCapsFontMask,
    NSPosterFontMask,
    NSCompressedFontMask,
    NSFixedPitchFontMask,
    NSUnitalicFontMask
};
```

DESCRIPTION These constants are used by the NSFontManager to identify font traits. Some traits are mutually exclusive, such as NSExpandedFontMask and NSCondensedFontMask.

NSFontPanel—Tags for Subviews

DECLARED IN AppKit/NSFontPanel.h

SYNOPSIS

```
enum {
    NSFPPreviewButton,
    NSFPRevertButton,
    NSFPSetButton,
    NSFPPreviewField,
    NSFPSizeField,
    NSFPSizeTitle,
    NSFPCurrentField
};
```

DESCRIPTION These tags identify the NSViews within an NSFontPanel.

NSGlyph—Attributes

DECLARED IN AppKit/NSLayoutManager.h

SYNOPSIS

```
enum _NSGlyphAttribute {
    NSGlyphAttributeSoft,
    NSGlyphAttributeElastic,
    NSGlyphAttributeInscribe
};
```

DESCRIPTION These glyph attributes are used only inside the glyph generation machinery, but must be shared between components.

NSGlyph—Reserved Glyph Codes

DECLARED IN AppKit/NSFont.h

SYNOPSIS

```
enum {
    NSControlGlyph = 0x00FFFFFF,
    NSNullGlyph = 0x0
};
```

DESCRIPTION These two values are reserved for the two named NSGlyphs.

NSImageRep—Display Device Matching

DECLARED IN AppKit/NSImageRep.h

SYNOPSIS

```
enum {
    NSImageRepMatchesDevice
};
```

DESCRIPTION This constant is used by NSImageRep to indicate that the value of certain attributes, such as the number of colors, or bits-per-sample, will change to match the display device. See the NSImageRep class specification for more information.

NSPageLayoutPanel—Tags for Controls

DECLARED IN AppKit/NSPageLayout.h

SYNOPSIS

```
enum {
    NSPLImageButton,
    NSPLTitleField,
    NSPLPaperNameButton,
    NSPLUnitsButton,
    NSPLWidthForm,
    NSPLHeightForm,
    NSPLOrientationMatrix,
    NSPLCancelButton,
    NSPLOKButton
};
```

DESCRIPTION These constants represent the tag values of the controls displayed by an NSPageLayoutPanel.

NSPanel—Alert Panel Return Values

DECLARED IN AppKit/NSPanel.h

SYNOPSIS

```
enum {
    NSAlertDefaultReturn,
    NSAlertAlternateReturn,
    NSAlertOtherReturn,
    NSAlertErrorReturn
};
```

DESCRIPTION These constants define values returned by the **NSRunAlertPanel** function and by the NSApplication method **runModalSession:** when the modal session is run with an NSPanel provided by the **NSGetAlertPanel** function.

NSPanel—Modal Panel Return Values

DECLARED IN AppKit/NSPanel.h

SYNOPSIS

```
enum {
    NSOKButton,
    NSCancelButton
};
```

DESCRIPTION These are the possible return values for such methods as the **runModal...** methods of NSOpenPanel, which tells which button (OK or Cancel) the user has clicked on an open panel. For other uses of these return values, see the class descriptions for NSPageLayout, NSPrintPanel and NSSavePanel.

NSPrintPanel—Tags for Subviews

DECLARED IN AppKit/NSPrintPanel.h

SYNOPSIS

```
enum {
    NSPPSaveButton,
    NSPPPreviewButton,
    NSFaxButton,
    NSPPTitleField,
    NSPPImageButton,
    NSPPNameTitle,
    NSPPNameField,
    NSPPNoteTitle,
    NSPPNoteField,
    NSPPStatusTitle,
    NSPPStatusField,
    NSPPCopiesField,
    NSPPPageChoiceMatrix,
    NSPPPageRangeFrom,
    NSPPPageRangeTo,
    NSPPScaleField,
    NSPPOptionsButton,
    NSPPPaperFeedButton,
    NSPPLayoutButton
};
```

DESCRIPTION These constants define tags for identifying the NSViews in a print panel in environments other than Microsoft Windows. Windows has its own way of handling print panels.

NSRunLoop—Ordering Modes for NSApplication

DECLARED IN AppKit/NSApplication.h

SYNOPSIS

```
enum {
    NSUpdateWindowsRunLoopOrdering
};
```

DESCRIPTION This constants is used with NSRunLoop's **performSelector:target:argument:order:modes:** method.

NSRunLoop—Ordering Mode for NSDPSServerContext

DECLARED IN AppKit/NSDPSServerContext.h

SYNOPSIS

```
enum {
    DPSFlushContextRunLoopOrdering
};
```

DESCRIPTION This constants is used with NSRunLoop's method **performSelector:target:argument:order:modes:**.

NSRunLoop—Ordering Modes for NSWindow

DECLARED IN AppKit/NSWindow.h

SYNOPSIS

```
enum {
    NSDisplayWindowRunLoopOrdering,
    NSResetCursorRectsRunLoopOrdering
};
```

DESCRIPTION These constants are passed to NSRunLoop’s method **performSelector:target:argument:order:modes:**.

NSSavePanel—Tags for Subviews

DECLARED IN AppKit/NSSavePanel.h

SYNOPSIS

```
enum {
    NSFileHandlingPanelImageButton,
    NSFileHandlingPanelTitleField,
    NSFileHandlingPanelBrowser,
    NSFileHandlingPanelCancelButton,
    NSFileHandlingPanelOKButton,
    NSFileHandlingPanelForm,
    NSFileHandlingPanelHomeButton,
    NSFileHandlingPanelDiskButton,
    NSFileHandlingPanelDiskEjectButton
};
```

DESCRIPTION These constants define tags for identifying NSViews in an NSSavePanel.

NSTextAttachment—Attachment Character

DECLARED IN AppKit/NSTextAttachment.h

SYNOPSIS

```
enum {
    NSAttachmentCharacter = 0xfffc
};
```

DECLARED IN This Unicode indicates the presence of an attachment in an NSAttributedString. For more information, see the Class Cluster Description of NSAttributedStringAdditions.

NSText—Important Unicodes

DECLARED IN AppKit/NSText.h

SYNOPSIS

```
enum {
    NSParagraphSeparatorCharacter = 0x2029,
    NSLineSeparatorCharacter = 0x2028,
    NSTabCharacter = 0x0009,
    NSFormFeedCharacter = 0x000c,
    NSNewlineCharacter = 0x000a,
    NSCarriageReturnCharacter = 0x000d,
    NSEnterCharacter = 0x0003,
    NSBackspaceCharacter = 0x0008,
    NSBackTabCharacter = 0x0019,
    NSDeleteCharacter = 0x007f,
};
```

DESCRIPTION These constants specify several commonly-used Unicode characters.

NSText—Movement Codes

DECLARED IN AppKit/NSText.h

SYNOPSIS

```
enum {
    NSIllegalTextMovement,
    NSReturnTextMovement,
    NSTabTextMovement,
    NSBacktabTextMovement,
    NSLeftTextMovement,
    NSRightTextMovement,
    NSUpTextMovement,
    NSDownTextMovement
};
```

DESCRIPTION These constants are the codes for movement between fields. They are the possible int values for the NSTextMovement key of NSTextDidEndEditingNotification. For more information, see the "Notifications" section of the NSText class specification.

NSTextStorage—Editing

DECLARED IN AppKit/NSTextStorage.h

SYNOPSIS

```
enum {
    NSTextStorageEditedAttributes,
    NSTextStorageEditedCharacters
};
```

DESCRIPTION These values, which may be combined by a bitwise OR, help describe the changes that an editing session has made to an NSTextStorage object. They are the return values of the NSTextStorage method **editedMask**, and the parameter values for the second slot of the NSLayoutManager method **textStorage:edited:...**.

NSView—Resizing

DECLARED IN AppKit/NSView.h

SYNOPSIS

```
enum {
    NSViewNotSizable,
    NSViewMinXMargin,
    NSViewWidthSizable,
    NSViewMaxXMargin,
    NSViewMinYMargin,
    NSViewHeightSizable,
    NSViewMaxYMargin
};
```

DESCRIPTION Used to describe which parts of an NSView (or its margins) are resized when the NSView's superNSView is resized. See the NSView class specification for details.

NSWindow—Border Masks

DECLARED IN AppKit/NSWindow.h

SYNOPSIS

```
enum {
    NSBorderlessWindowMask,
    NSTitledWindowMask,
    NSClosableWindowMask,
    NSMiniaturizableWindowMask,
    NSResizableWindowMask
};
```

DESCRIPTION These determine the presence of a title and various buttons in an NSWindow's border.

NSWindow—Window Levels

DECLARED IN AppKit/NSWindow.h

SYNOPSIS

```
enum {
    NSNormalWindowLevel,
    NSFloatingWindowLevel,
    NSDockWindowLevel,
    NSSubmenuWindowLevel,
    NSTornOffMenuWindowLevel,
    NSMainMenuWindowLevel,
    NSModalPanelWindowLevel,
    NSPopUpMenuWindowLevel
};
```

DESCRIPTION These constants name the Application Kit's window levels. The stacking of levels takes precedence over the stacking of windows within each level. That is, even the bottom window in a level will obscure even the top window of the next level down.

The constant NSTornOffMenuWindowLevel is preferable to its synonym, NSSubmenuWindowLevel.

Global Variables

Application Kit—Exceptions

DECLARED IN AppKit/NSErrors.h

SYNOPSIS

NSString ***NSTextLineTooLongException**;
NSString ***NSTextNoSelectionException**;
NSString ***NSWordTablesWriteException**;
NSString ***NSWordTablesReadException**;
NSString ***NSTextReadException**;
NSString ***NSTextWriteException**;
NSString ***NSPasteboardCommunicationException**;
NSString ***NSPrintingCommunicationException**;
NSString ***NSAbortModalException**;
NSString ***NSAbortPrintingException**;
NSString ***NSIllegalSelectorException**;
NSString ***NSAppKitVirtualMemoryException**;
NSString ***NSBadRTFDirectiveException**;
NSString ***NSBadRTFFontTableException**;
NSString ***NSBadRTFStyleSheetException**;
NSString ***NSTypedStreamVersionException**;
NSString ***NSTIFFException**;
NSString ***NSPrintPackageException**;
NSString ***NSBadRTFColorTableException**;
NSString ***NSDraggingException**;

NSString ***NSColorListIOException**;
NSString ***NSColorListNotEditableException**;
NSString ***NSBadBitmapParametersException**;
NSString ***NSWindowServerCommunicationException**;
NSString ***NSFontUnavailableException**;
NSString ***NSPPDIncludeNotFoundException**;
NSString ***NSPPDParseException**;
NSString ***NSPPDIncludeStackOverflowException**;
NSString ***NSPPDIncludeStackUnderflowException**;
NSString ***NSRTFPropertyStackOverflowException**;
NSString ***NSAppKitIgnoredException**;
NSString ***NSBadComparisonException**;
NSString ***NSImageCacheException**;
NSString ***NSNibLoadingException**;
NSString ***NSBrowserIllegalDelegateException**;

DESCRIPTION These constants name the exceptions that the Application Kit can raise.

Display Device—Descriptions

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS
NSString ***NSDeviceResolution**;
NSString ***NSDeviceColorSpaceName**;
NSString ***NSDeviceBitsPerSample**;
NSString ***NSDeviceIsScreen**;
NSString ***NSDeviceIsPrinter**;
NSString ***NSDeviceSize**;

DESCRIPTION These are the keys for device description dictionaries, such as those returned by the **deviceDictionary** methods of NSPrinter, NSScreen and NSWindow.

NSDeviceResolution is an NSValue containing an NSSize in dots per inch. NSColorSpaceName is an NSString describing the color space of the device. NSDeviceBitsPerSample is an NSValue containing an int. NSDeviceIsScreen and NSDeviceIsPrinter are boolean values that tell whether the device is a screen or a printer. NSDeviceSize is an NSValue containing an NSSize that represents the device's size in points.

NSApplication—Notifications

DECLARED IN AppKit/NSApplication.h

SYNOPSIS

NSString ***NSApplicationDidBecomeActiveNotification**;
NSString ***NSApplicationDidFinishLaunchingNotification**;
NSString ***NSApplicationDidHideNotification**;
NSString ***NSApplicationDidResignActiveNotification**;
NSString ***NSApplicationDidUnhideNotification**;
NSString ***NSApplicationDidUpdateNotification**;
NSString ***NSApplicationWillBecomeActiveNotification**;
NSString ***NSApplicationWillFinishLaunchingNotification**;
NSString ***NSApplicationWillHideNotification**;
NSString ***NSApplicationWillResignActiveNotification**;
NSString ***NSApplicationWillUnhideNotification**;
NSString ***NSApplicationWillUpdateNotification**;
NSString ***NSApplicationWillTerminateNotification**;

DECLARED IN These are the notifications used with the methods of the NSApplicationNotifications category of NSObject.

NSApplication—Shared Application Object

DECLARED IN AppKit/NSApplication.h

SYNOPSIS id **NSApp**;

DESCRIPTION This variable designates the shared application object, created by NSApplication's **sharedApplication** method.

NSAttributedString—Attributes

DECLARED IN AppKit/NSAttributedString.h

SYNOPSIS

NSString ***NSFontAttributeName**;
NSString ***NSParagraphStyleAttributeName**;
NSString ***NSForegroundColorAttributeName**;
NSString ***NSUnderlineStyleAttributeName**;
NSString ***NSSuperscriptAttributeName**;
NSString ***NSBackgroundColorAttributeName**;
NSString ***NSAttachmentAttributeName**;
NSString ***NSLigatureAttributeName**;
NSString ***NSBaselineOffsetAttributeName**;
NSString ***NSKernAttributeName**;

DESCRIPTION These strings define the supported attributes of NSAttributedStrings. For more information, see the “Accessing Attributes” section in the NSAttributedString class cluster specification.

NSComboBox—Notifications

DECLARED IN AppKit/NSComboBox.h

SYNOPSIS

NSString ***NSComboBoxWillPopUpNotification**;
NSString ***NSComboBoxWillDismissNotification**;
NSString ***NSComboBoxSelectionDidChangeNotification**;
NSString ***NSComboBoxSelectionIsChangingNotification**;

DESCRIPTION These notifications are sent by NSComboBoxes.

NSColor—Color Space Names

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS

NSString ***NSCalibratedWhiteColorSpace**;
NSString ***NSCalibratedBlackColorSpace**;
NSString ***NSCalibratedRGBColorSpace**;
NSString ***NSDeviceWhiteColorSpace**;
NSString ***NSDeviceBlackColorSpace**;
NSString ***NSDeviceRGBColorSpace**;
NSString ***NSDeviceCMYKColorSpace**;
NSString ***NSNamedColorSpace**;
NSString ***NSCustomColorSpace**;

DESCRIPTION These are the predefined names for color spaces. In the two ...WhiteColorSpaces, white corresponds to a value of 1.0. In the two ...BlackColorSpaces, black corresponds to a value of 1.0. NSNamedColorSpace is used for “catalog” colors—that is, colors specified by names rather than coordinates. NSCustomColorSpace indicates a custom color space, which can be useful in working with images; unlike the other color spaces, NSCustomColorSpace is not used with NSColors.

NSColor—Grayscale Values

DECLARED IN AppKit/NSGraphics.h

SYNOPSIS

const float **NSWhite**;
const float **NSLightGray**;
const float **NSDarkGray**;
const float **NSBlack**;

DESCRIPTION These are the standard gray values for the 2-bit deep grayscale color space.

NSColor—Notifications

DECLARED IN AppKit/NSColor.h

SYNOPSIS NSString ***NSSystemColorsDidChangeNotification**;

DESCRIPTION This notification is sent when the system colors have been changed (such as through a system control panel interface). For more on system colors, see the “System Colors” section of the NSColor class specification.

NSColorList—Notifications

DECLARED IN AppKit/NSColorList.h

SYNOPSIS NSString ***NSColorListDidChangeNotification**;

DESCRIPTION When an NSColorList changes, it posts this notification.

NSColorPanel—Notifications

DECLARED IN AppKit/NSColorPanel.h

SYNOPSIS NSString ***NSColorPanelColorDidChangeNotification**;

DESCRIPTION When an NSColorPanel changes, it posts this notification.

NSControl—Notifications

DECLARED IN AppKit/NSControl.h

SYNOPSIS NSString ***NSControlTextDidBeginEditingNotification**;
NSString ***NSControlTextDidEndEditingNotification**;
NSString ***NSControlTextDidChangeNotification**;

DESCRIPTION NSControls containing editable text can send these notifications. For more information, see the "Notifications" section of the NSControl class specification.

NSDataLink—Filename Extension

DECLARED IN AppKit/NSDataLink.h

SYNOPSIS NSString ***NSDataLinkFilenameExtension**;

DESCRIPTION NSDataLinkFilenameExtension is the filename extension used for links saved to files using NSDataLink's **saveLinkIn:** or **writeToFile:** methods.

NSFont—Keys to the AFM Dictionary

DECLARED IN AppKit/NSFont.h

SYNOPSIS

NSString ***NSAFMFamilyName**;
NSString ***NSAFMFontName**;
NSString ***NSAFMFormatVersion**;
NSString ***NSAFMFullName**;
NSString ***NSAFMNotice**;
NSString ***NSAFMVersion**;
NSString ***NSAFMWeight**;
NSString ***NSAFMEncodingScheme**;
NSString ***NSAFMCharacterSet**;
NSString ***NSAFMCapHeight**;
NSString ***NSAFMXHeight**;
NSString ***NSAFMAscender**;
NSString ***NSAFMDescender**;
NSString ***NSAFMUnderlinePosition**;
NSString ***NSAFMUnderlineThickness**;
NSString ***NSAFMItalicAngle**;
NSString ***NSAFMMappingScheme**;

DESCRIPTION These are the keys to the font information dictionary returned by NSFont's **afmDictionary** method. To convert values like NSAFMCapHeight to floats, use NSString's **floatValue** method.

For other font information, use NSFont's **afmFileContents** method.

NSFont—PostScript Transformation Matrix

DECLARED IN AppKit/NSFont.h

SYNOPSIS

const float ***NSFontIdentityMatrix**;

DESCRIPTION NSFontIdentityMatrix is a PostScript transformation matrix useful as a parameter to the NSFont method **fontWithName:matrix:**.

NSHelpManager—Notifications

DECLARED IN AppKit/NSHelpManager.h

SYNOPSIS NSString ***NSContextHelpModeDidActivateNotification**;
NSString ***NSContextHelpModeDidDeactivateNotification**;

DESCRIPTION These are notifications for the activation and deactivation of the context help mode.

NSImageRep—Notifications

DECLARED IN AppKit/NSImageRep.h

SYNOPSIS NSString ***NSImageRepRegistryDidChangeNotification**;

DESCRIPTION This notification is sent when the NSImageRep class registry changes.

NSInterfaceStyleDefault

DECLARED IN AppKit/NSInterfaceStyle.h

SYNOPSIS NSString ***NSInterfaceStyleDefault**;

DESCRIPTION NSInterfaceStyleDefault can be used to override the platform's default interface style. For more information, see the function **NSInterfaceStyleForKey**.

NSPasteboard—Names

DECLARED IN AppKit/NSPasteboard.h

SYNOPSIS
NSString ***NSGeneralPboard**;
NSString ***NSFontPboard**;
NSString ***NSRulerPboard**;
NSString ***NSFindPboard**;
NSString ***NSDragPboard**;

DESCRIPTION Some standard pasteboard names. See the NSPasteboard class specification for more information.

NSPasteboard—Type for Data Links

DECLARED IN AppKit/NSDataLink.h

SYNOPSIS NSString ***NSDataLinkPboardType**;

DESCRIPTION A pasteboard type for copying a data link to the pasteboard. See the NSDataLink class specification for more information.

NSPasteboard—Type for Selection Descriptions

DECLARED IN AppKit/NSSelection.h

SYNOPSIS NSString ***NSSelectionPboardType**;

DESCRIPTION A pasteboard type for copying selection descriptions to the pasteboard. See the NSSelection class specification for more information.

NSPasteboard—Types for Standard Data

DECLARED IN AppKit/NSPasteboard.h

SYNOPSIS

NSString ***NSStringPboardType**;
NSString ***NSFilenamesPboardType**;
NSString ***NSPostScriptPboardType**;
NSString ***NSTIFFPboardType**;
NSString ***NSRTFPboardType**;
NSString ***NSTabularTextPboardType**;
NSString ***NSFontPboardType**;
NSString ***NSRulerPboardType**;
NSString ***NSFileContentsPboardType**;
NSString ***NSColorPboardType**;
NSString ***NSRTFDPboardType**;

DESCRIPTION Some standard pasteboard data types. See the NSPasteboard class specification for more information.

NSPrintInfo—Dictionary Keys

DECLARED IN AppKit/NSPrintInfo.h

SYNOPSIS

NSString ***NSPrintPaperName**;
NSString ***NSPrintPaperSize**;
NSString ***NSPrintFormName**;
NSString ***NSPrintMustCollate**;
NSString ***NSPrintOrientation**;
NSString ***NSPrintLeftMargin**;
NSString ***NSPrintRightMargin**;
NSString ***NSPrintTopMargin**;
NSString ***NSPrintBottomMargin**;
NSString ***NSPrintHorizontallyCentered**;
NSString ***NSPrintVerticallyCentered**;
NSString ***NSPrintHorizontalPagination**;
NSString ***NSPrintVerticalPagination**;
NSString ***NSPrintScalingFactor**;
NSString ***NSPrintAllPages**;
NSString ***NSPrintReversePageOrder**;
NSString ***NSPrintFirstPage**;
NSString ***NSPrintLastPage**;
NSString ***NSPrintCopies**;
NSString ***NSPrintPagesPerSheet**;
NSString ***NSPrintJobFeatures**;
NSString ***NSPrintPaperFeed**;
NSString ***NSPrintManualFeed**;
NSString ***NSPrintPrinter**;
NSString ***NSPrintJobDisposition**;

NSString ***NSPrintSavePath**;
NSString ***NSPrintFaxReceiverNames**;
NSString ***NSPrintFaxReceiverNumbers**;
NSString ***NSPrintFaxSendTime**;
NSString ***NSPrintFaxUseCoverSheet**;
NSString ***NSPrintFaxCoverSheetName**;
NSString ***NSPrintFaxReturnReceipt**;
NSString ***NSPrintFaxHighResolution**;
NSString ***NSPrintFaxTrimPageEnds**;
NSString ***NSPrintFaxModem**;
NSString ***NSPrintSpoolJob**;
NSString ***NSPrintFaxJob**;
NSString ***NSPrintPreviewJob**;
NSString ***NSPrintSaveJob**;
NSString ***NSPrintCancelJob**;

DESCRIPTION These are the keys to the NSPrintInfo NSDictionary. For a table explaining them, see the NSPrintInfo method **initWithDictionary:**.

NSPopUpButton—Notification

DECLARED IN AppKit/NSPopUpButton.h

SYNOPSIS NSString ***NSPopUpButtonWillPopUpNotification**;

DESCRIPTION NSPopUpButton sends this notification when an instance of it is about to pop up.

NSPrintOperation—Exception

DECLARED IN AppKit/NSPrintOperation.h

SYNOPSIS NSString ***NSPrintOperationExistsException**;

DESCRIPTION This exception is raised when there is already a print operation in process. The methods that raise it are the **EPSOperation...** and **printOperation...** methods in NSPrintOperation:

NSRunLoop—Modes

DECLARED IN AppKit/NSApplication.h

SYNOPSIS NSString ***NSModalPanelRunLoopMode**;
NSString ***NSEventTrackingRunLoopMode**;

DESCRIPTION These are modes passed to NSRunLoop

NSSplitView—Notifications

DECLARED IN AppKit/NSSplitView.h

SYNOPSIS NSString ***NSSplitViewDidResizeSubviewsNotification**;
NSString ***NSSplitViewWillResizeSubviewsNotification**;

DESCRIPTION These are the notifications that an NSSplitView can send.

NSTableView—Notifications

DECLARED IN AppKit/NSTableView.h

SYNOPSIS NSString ***NSTableViewSelectionDidChangeNotification**;
NSString ***NSTableViewColumnDidMoveNotification**;
NSString ***NSTableViewColumnDidResizeNotification**;
NSString ***NSTableViewSelectionIsChangingNotification**;

DESCRIPTION These are the notifications that an NSTableView can send.

NSText—Notifications

DECLARED IN AppKit/NSText.h

SYNOPSIS NSString ***NSTextDidBeginEditingNotification**;
NSString ***NSTextDidEndEditingNotification**;
NSString ***NSTextDidChangeNotification**;

DESCRIPTION These notifications can be sent by an NSText object. For explanations, see the “Notifications” section of the NSText class specification.

NSTextStorage—Notifications

DECLARED IN AppKit/NSTextStorage.h

SYNOPSIS NSString ***NSTextStorageWillProcessEditingNotification**;
NSString ***NSTextStorageDidProcessEditingNotification**;

DESCRIPTION These notifications can be sent by an NSTextStorage object. For explanations, see the “Notifications” section of the NSTextStorage class specification.

NSTextView—Notifications

DECLARED IN AppKit/NSTextView.h

SYNOPSIS NSString ***NSTextViewWillChangeNotifyingTextViewNotification**;
NSString ***NSTextViewDidChangeSelectionNotification**;

DESCRIPTION These notifications can be sent by an NSTextView object. For explanations, see the “Notifications” section of the NSTextView class specification.

The notifications that NSTextView most often sends are the ones that it inherits from NSText.

NSView—Notifications

DECLARED IN AppKit/NSView.h

SYNOPSIS
NSString ***NSViewFrameDidChangeNotification**;
NSString ***NSViewFocusDidChangeNotification**;
NSString ***NSViewBoundsDidChangeNotification**;

DESCRIPTION These notifications are sent by NSViews.

The last notification, NSViewBoundsDidChangeNotification, is sent when the view bounds change but the frame does not. That is, it is sent whenever the view's bounds are translated, scaled or rotated, but not when the bounds change in response to, say, a **setFrameSize:** message.

NSWindow—Notifications

DECLARED IN AppKit/NSWindow.h

SYNOPSIS
NSString ***NSWindowDidBecomeKeyNotification**;
NSString ***NSWindowDidBecomeMainNotification**;
NSString ***NSWindowDidChangeScreenNotification**;
NSString ***NSWindowDidDeminiaturizeNotification**;
NSString ***NSWindowDidExposeNotification**;
NSString ***NSWindowDidMiniaturizeNotification**;
NSString ***NSWindowDidMoveNotification**;
NSString ***NSWindowDidResignKeyNotification**;
NSString ***NSWindowDidResignMainNotification**;
NSString ***NSWindowDidResizeNotification**;
NSString ***NSWindowDidUpdateNotification**;
NSString ***NSWindowWillCloseNotification**;
NSString ***NSWindowWillMiniaturizeNotification**;
NSString ***NSWindowWillMoveNotification**;

DESCRIPTION These are the notifications that can be sent by an NSWindow object. For explanations, see the “Notifications” section of the NSWindow class specification.

NSWindow—Sizes

DECLARED IN AppKit/NSWindow.h

SYNOPSIS

NSSize **NSIconSize**;
NSSize **NSTokenSize**;

DESCRIPTION On some platforms, a token is a bezeled tile used to represent a docked application or a miniaturized document, and an icon is the image drawn inside a token.

On platforms that support tokens and icons, these size constants can be used for drawing inside them. It is more portable, however, to change an icon by using the NSApplication method **setApplicationIconImage:** or the NSWindow method **setMiniwindowImage:**.

NSWorkspace—File Operation Constants

DECLARED IN AppKit/NSWorkspace.h

SYNOPSIS

NSString ***NSWorkspaceMoveOperation**;
NSString ***NSWorkspaceCopyOperation**;
NSString ***NSWorkspaceLinkOperation**;
NSString ***NSWorkspaceCompressOperation**;
NSString ***NSWorkspaceDecompressOperation**;
NSString ***NSWorkspaceEncryptOperation**;
NSString ***NSWorkspaceDecryptOperation**;
NSString ***NSWorkspaceDestroyOperation**;
NSString ***NSWorkspaceRecycleOperation**;
NSString ***NSWorkspaceDuplicateOperation**;

DESCRIPTION These constants define possible values for the *operation* slot in NSWorkspace's **performFileOperation:**... method.

NSWorkspace—File Types

DECLARED IN AppKit/NSWorkspace.h

SYNOPSIS
NSString ***NSPlainFileType**;
NSString ***NSDirectoryFileType**;
NSString ***NSApplicationFileType**;
NSString ***NSFilesystemFileType**;
NSString ***NSShellCommandFileType**;

DESCRIPTION These values are used in the final parameter slot of the NSWorkspace method **getInfoForFile: application:type:**.

NSWorkspace—Notifications

DECLARED IN AppKit/NSWorkspace.h

SYNOPSIS
NSString ***NSWorkspaceDidLaunchApplicationNotification**;
NSString ***NSWorkspaceDidMountNotification**;
NSString ***NSWorkspaceDidPerformFileOperationNotification**;
NSString ***NSWorkspaceDidTerminateApplicationNotification**;
NSString ***NSWorkspaceDidUnmountNotification**;
NSString ***NSWorkspaceWillLaunchApplicationNotification**;
NSString ***NSWorkspaceWillPowerOffNotification**;
NSString ***NSWorkspaceWillUnmountNotification**;

DESCRIPTION These notifications come through the special notification center. For more information, see the "Notifications" section of the NSWorkspace class specification.

NSChangeSpelling

Adopted By: NSText

Declared In: AppKit/NSSpellProtocol.h

Protocol Description

This protocol is implemented by objects in the responder chain that can correct a misspelled word. See the description of the NSSpellChecker class for more information.

Instance Methods

changeSpelling:

– (void)**changeSpelling:**(id)*sender*

Implement this method to replace the selected word in the receiver with a corrected version from the Spelling panel. This message is sent by the NSSpellChecker to the object whose text is being checked. To get the corrected spelling, ask the sender for the string value of its selected cell (visible to the user as the text field in the Spelling panel). This method should replace the selected portion of the text with the string that it gets from the NSSpellChecker.

NSColorPickingCustom

Adopted By: NSColorPicker

Declared In: AppKit/NSColorPicking.h

Protocol Description

Together with the NSColorPickingDefault protocol, NSColorPickingCustom provides a way to add color pickers—custom user interfaces for color selection—to an application's NSColorPanel. The NSColorPickingDefault protocol provides basic behavior for a color picker. The NSColorPicker class adopts the NSColorPickingDefault protocol. The easiest way to implement a color picker is to create a subclass of NSColorPicker and implement the NSColorPickingCustom protocol for this new class.

Note: All of NSColorPickingCustom's methods must be implemented by the custom color picker.

Here are the standard color picking modes and mode constants (defined in **AppKit/NSColorPanel.h**):

Mode	Color Mode Constant
Grayscale-Alpha	NSGrayModeColorPanel
Red-Green-Blue	NSRGBModeColorPanel
Cyan-Yellow-Magenta-Black	NSCMYKModeColorPanel
Hue-Saturation-Brightness	NSHSBModeColorPanel
Custom palette	NSCustomPaletteModeColorPanel
Custom color list	NSColorListModeColorPanel
Color wheel	NSWheelModeColorPanel

In grayscale-alpha, red-green-blue, cyan-magenta-yellow-black, and hue-saturation-brightness modes, the user adjusts colors by manipulating sliders. In the custom palette mode, the user can load an NSImage file (TIFF or EPS) into the NSColorPanel, then select colors from the image. In custom color list mode, the user can create and load lists of named colors. The two custom modes provide NSPopUpLists for loading and saving files. Finally, color wheel mode provides a simplified control for selecting colors.

If your color picker includes submodes, you should define a unique value for each submode. As an example, the slider picker has four values defined in the above list (NSGrayModeColorPanel,

NSRGBModeColorPanel, NSCMYKModeColorPanel, and NSHSBModeColorPanel)—one for each of its submodes.

Method Types

Setting the Current Color

– setColor:

Getting the Mode

– currentMode
– supportsMode:

Getting the View

– provideNewView:

Instance Methods

currentMode

– (int)**currentMode**

Returns the color picker's current mode (or submode, if applicable). The returned value should be unique to your color picker. See this protocol description's list of the unique values for the standard color pickers used by the Application Kit.

See also: **– supportsMode:**

provideNewView:

– (NSView *)**provideNewView:**(BOOL)*initialRequest*

Returns the view containing the color picker's user interface. This message is sent to the color picker whenever the color panel attempts to display it. This may be when the panel is first presented, when the user switches pickers, or when the picker is switched through API. The argument *initialRequest* is YES only when this method is first invoked for your color picker. If *initialRequest* is YES, the method should perform any initialization required (such as lazily loading a nib file, initializing the view, or performing any other custom initialization required for your picker). The NSView returned by this method should be set to automatically resize both its width and height.

setColor:

– (void)**setColor:**(NSColor *)*color*

Adjusts the color picker to make *color* the currently selected color. This method is invoked on the current color picker each time NSColorPanel's **setColor:** method is invoked. If *color* is actually different from the color picker's color (as it would be if, for example, the user dragged a color into NSColorPanel's color well), this method could be used to update the color picker's color to reflect the change.

supportsMode:

– (BOOL)**supportsMode:**(int)*mode*

Returns whether or not the receiver supports the specified picking mode. This method is invoked when the NSColorPanel's is first initialized: It is used to attempt to restore the user's previously selected mode. It is also invoked by NSColorPanel's **setMode:** method to find the color picker that supports a particular mode. See this protocol description's list of the unique mode values for the standard color pickers used by the Application Kit.

See also: **– currentMode**

NSColorPickingDefault

Adopted By: NSColorPicker

Declared In: AppKit/NSColorPicking.h

Protocol Description

The NSColorPickingDefault protocol, together with the NSColorPickingCustom protocol, provides an interface for adding color pickers—custom user interfaces for color selection—to an application's NSColorPanel. The NSColorPickingDefault protocol provides basic behavior for a color picker. The NSColorPickingCustom protocol provides implementation-specific behavior.

The NSColorPicker class implements the NSColorPickingDefault protocol. The simplest way to implement your own color picker is to create a subclass of NSColorPicker, implementing the NSColorPickingCustom protocol for that subclass. However, it's possible to create a subclass of another class, such as NSView, and use it as a base upon which to add the methods of both NSColorPickingDefault and NSColorPickingCustom.

Color Picker Bundles

A class that implements the NSColorPickingDefault and NSColorPickingCustom protocols needs to be compiled and linked in an application's object file. However, your application need not explicitly create an instance of this class. Instead, your application's file package should include a directory named **ColorPickers**; within this directory you should place a directory *MyPickerClass***.bundle** for each custom color picker your application implements. This bundle should contain all resources required for your color picker: nib files, TIFF files, and so on.

NSColorPanel will allocate and initialize an instance of each class for which a bundle is found in the **ColorPickers** directory. The class name is assumed to be the bundle directory name minus the **.bundle** extension.

Color Picker Buttons

NSColorPanel lets the user select a color picker from an NSMatrix of NSButtonCells. This protocol includes methods for providing and manipulating the image that gets displayed on the button.

Color Mask and Color Modes

The color mask determines which color mode is enabled for NSColorPanel. This mask is set before you initialize a new instance of NSColorPanel. NSColorPanelAllModesMask represents the logical OR of the

other color mask constants: It causes the NSColorPanel to display all standard color pickers. When initializing a new instance of NSColorPanel, you can logically OR any combination of color mask constants to restrict the available color modes. The predefined color mask constants are:

Mode	Color Mask Constant
Grayscale-Alpha	NSColorPanelGrayModeMask
Red-Green-Blue	NSColorPanelRGBModeMask
Cyan-Yellow-Magenta-Black	NSColorPanelCMYKModeMask
Hue-Saturation-Brightness	NSColorPanelHSBModeMask
Custom palette	NSColorPanelCustomPaletteModeMask
Custom color list	NSColorPanelColorListModeMask
Color wheel	NSColorPanelWheelModeMask
All of the above	NSColorPanelAllModesMask

When an application's instance of NSColorPanel is masked for more than one color mode, your program can set its active mode by invoking the **setMode:** method with a color mode constant as its argument; the user can set the mode by clicking buttons on the panel. Here are the standard color modes and mode constants:

Mode	Color Mode Constant
Grayscale-Alpha	NSGrayModeColorPanel
Red-Green-Blue	NSRGBModeColorPanel
Cyan-Yellow-Magenta-Black	NSCMYKModeColorPanel
Hue-Saturation-Brightness	NSHSBModeColorPanel
Custom palette	NSCustomPaletteModeColorPanel
Custom color list	NSColorListModeColorPanel
Color wheel	NSWheelModeColorPanel

In grayscale-alpha, red-green-blue, cyan-magenta-yellow-black, and hue-saturation-brightness modes, the user adjusts colors by manipulating sliders. In the custom palette mode, the user can load an NSImage file (TIFF or EPS) into the NSColorPanel, then select colors from the image. In custom color list mode, the user can create and load lists of named colors. The two custom modes provide NSPopUpLists for loading and saving files. Finally, color wheel mode provides a simplified control for selecting colors.

These constants are defined in **AppKit/NSColorPanel.h**.

Method Types

Initializing a Color Picker
– initWithPickerMask:colorPanel:

Setting the Mode
– setMode:

Using Color Lists
– attachColorList:
– detachColorList:

Adding Button Images
– insertNewButtonImage:in:
– provideNewButtonImage

Showing Opacity Controls
– alphaControlAddedOrRemoved:

Responding to a Resized View
– viewSizeChanged:

Instance Methods

alphaControlAddedOrRemoved:

– (void)**alphaControlAddedOrRemoved:**(id)*sender*

Sent by the color panel when the opacity controls have been hidden or displayed. Invoked automatically when the NSColorPanel's opacity slider is added or removed; you never invoke this method directly.

If the color picker has its own opacity controls, it should hide or display them, depending on whether the sender's **showsAlpha** method returns NO or YES.

attachColorList:

– (void)**attachColorList:**(NSColorList *)*colorList*

Tells the color picker to attach the given *colorList*, if it isn't already displaying the list. You never invoke this method; it's invoked automatically by the NSColorPanel when its **attachColorList:** method is invoked. Since NSColorPanel's list mode manages NSColorLists, this method need only be implemented by a custom color picker that manages NSColorLists itself. This method ordinarily doesn't do anything, since NSColorPanel's list mode manages NSColorLists.

See also: **– detachColorList:**

detachColorList:

– (void)**detachColorList:**(NSColorList *)*colorList*

Tells the color picker to detach the given *colorList*, unless the receiver isn't displaying the list. You never invoke this method; it's invoked automatically by the NSColorPanel when its **detachColorList:** method is invoked. Since NSColorPanel's list mode manages NSColorLists, this method need only be implemented by a custom color picker that manages NSColorLists itself. This method ordinarily doesn't do anything, since NSColorPanel's list mode manages NSColorLists.

See also: **– attachColorList:**

initWithPickerMask:colorPanel:

– (id)**initWithPickerMask:**(int)*mask*
colorPanel:(NSColorPanel *)*owningColorPanel*

Notifies the color picker of the color panel's mask and initializes the color picker. This method is sent by the NSColorPanel to all implementors of the color picking protocols when the application's color panel is first initialized. In order for your color picker to receive this message, it must have a bundle in your application's "ColorPickers" directory (described in "Color Picker Bundles" in the Protocol Description).

mask is determined by the argument to the NSColorPanel method **setPickerMask:**. If no mask has been set, *mask* is NSColorPanelAllModesMask. If your color picker supports any additional modes, you should invoke the **setPickerMask:** method when your application initializes to notify the NSColorPanel class. The standard mask constants are:

Mode	Color Mask Constant
Grayscale-Alpha	NSColorPanelGrayModeMask
Red-Green-Blue	NSColorPanelRGBModeMask

Mode	Color Mask Constant
Cyan-Yellow-Magenta-Black	NSColorPanelCMYKModeMask
Hue-Saturation-Brightness	NSColorPanelHSBModeMask
Custom palette	NSColorPanelCustomPaletteModeMask
Custom color list	NSColorPanelColorListModeMask
Color wheel	NSColorPanelWheelModeMask
All of the above	NSColorPanelAllModesMask

This method should examine the mask and determine whether it supports any of the modes included there. You may also check the value in *mask* to enable or disable any subpickers or optional controls implemented by your color picker. Your color picker may also retain *owningColorPanel* in an instance variable for future communication with the color panel.

This method is provided to initialize your color picker; however, much of a color picker's initialization may be done lazily through the NSColorPickingCustom protocol's **provideNewView:** method. If your color picker responds to any of the modes represented in *mask*, it should perform its initialization and return **self**. Color pickers that do so have their buttons inserted in the color panel and continue to receive messages from the panel as the user manipulates it. If the color picker doesn't respond to any of the modes represented in mask, it should do nothing and return **nil**.

See also: **+ setPickerMask:** (NSColorPanel class)

insertNewButtonImage:in:

– (void)**insertNewButtonImage:**(NSImage *)*newButtonImage*
in:(NSButtonCell *)*buttonCell*

Sets *newButtonImage* as *buttonCell*'s image. *buttonCell* is the NSButtonCell object that lets the user choose the picker from the color panel—the color picker's representation in the NSColorPanel's picker NSMatrix. This method should perform application-specific manipulation of the image before it's inserted and displayed by the button cell.

See also: **– provideNewButtonImage**

provideNewButtonImage

– (NSImage *)**provideNewButtonImage**

Returns the image for the mode button that the user uses to select this picker in the color panel, that is, the color picker's representation in the NSColorPanel's picker NSMatrix. (This is the same image that the color panel uses as an argument when sending the **insertNewButtonImage:in:** message.)

setMode:

– (void)**setMode:**(int)*mode*

Sets the color picker's mode. This method is invoked by NSColorPanel's **setMode:** method to ensure that the color picker reflects the current mode. For example, invoke this method during color picker initialization to ensure that all color pickers are restored to the mode the user left them in the last time an NSColorPanel was used.

Most color pickers have only one mode, and thus don't need to do any work in this method. An example of a color picker that uses this method is the slider picker, which can choose from one of several submodes depending on the value of *mode*. The available modes are:

Mode	Color Mode Constant
Grayscale-Alpha	NSGrayModeColorPanel
Red-Green-Blue	NSRGBModeColorPanel
Cyan-Yellow-Magenta-Black	NSCMYKModeColorPanel
Hue-Saturation-Brightness	NSHSBModeColorPanel
Custom palette	NSCustomPaletteModeColorPanel
Custom color list	NSColorListModeColorPanel
Color wheel	NSWheelModeColorPanel

viewSizeChanged:

– (void)**viewSizeChanged:**(id)*sender*

Tells the color picker when the NSColorPanel's view size changes in a way that might affect the color picker. *sender* is the NSColorPanel that contains the color picker. Use this method to perform special preparation when resizing the color picker's view. Since this method is invoked only as appropriate, it's

better to implement this method than to override the method **superviewSizeChanged:** for the NSView in which the color picker's user interface is contained.

See also: **– provideNewView:** (NSColorPickingCustom protocol)

NSComboBoxCellDataSource

(Informal Protocol)

Category Of: NSObject

Declared In: AppKit/NSComboBoxCell.h

Category Description

The NSComboBoxCellDataSource category declares the methods that an NSComboBoxCell uses to access the contents of its data source object. The combo box cell determines how many items to display by sending a **numberOfItemsInComboBoxCell:** message, and accesses individual values with the **comboBoxCell: objectValueForItemAtIndex:** method. Incremental searches—performed when a user types into the combo box's text field while the pop-up list is displayed—are performed by sending **comboBoxCell: indexOfItemWithStringValue:** messages to the combo box cell's data source.

The NSComboBoxCell treats objects provided by its data source as values to be displayed in the combo box's pop-up list. If these objects aren't of common value classes—such as NSString, NSNumber, and so on—you'll need to create a custom NSFormatter to display them. See the NSFormatter class specification for more information.

When an NSComboBoxCellDataSource is asked to supply a data item, the NSComboBoxCell that sends the request is provided as a parameter. This allows a single data source object to manage several sets of data, choosing the appropriate set based on the identify of the NSComboBoxCell that sends the message.

Instance Methods

comboBoxCell:indexOfItemWithStringValue:

– (unsigned int)**comboBoxCell:**(NSComboBoxCell *)*aComboBoxCell*
indexOfItemWithStringValue:(NSString *)*aString*

An NSComboBoxCell uses this method to perform incremental—or "smart"—searched when the user types into the text field with the pop-up list displayed. Your implementation of this method should return the index for the item which matches *aString*, or NSNotFound if no item matches. This method is optional; if you don't provide an implementation for this method, no searches occur.

comboBoxCell:objectValueForItemAtIndex:

– (id)**comboBoxCell:**(NSComboBoxCell *)*aComboBoxCell* **objectValueForItemAtIndex:**(int)*index*

Implement this method to return the object that corresponds to the item at *index* in *aComboBoxCell*. Your data source must implement this method.

numberOfItemsInComboBoxCell:

– (int)**numberOfItemsInComboBoxCell:**(NSComboBoxCell *)*aComboBoxCell*

Implement this method to return the number of items managed for *aComboBoxCell* by your data source object. An NSComboBoxCell uses this method to determine how many items it should display in its pop-up list. Your data source must implement this method.

NSComboBoxDataSource

(Informal Protocol)

Category Of: NSObject

Declared In: AppKit/NSComboBox.h

Category Description

The NSComboBoxDataSource category declares the methods that an NSComboBox uses to access the contents of its data source object. The combo box determines how many items to display by sending a **numberOfItemsInComboBox:** message, and accesses individual values with the **comboBox: objectValueForItemAtIndex:** method. Incremental searches—performed when a user types into the combo box's text field while the pop-up list is displayed—are performed by sending **comboBox: indexOfItemWithStringValue:** messages to the combo box's data source.

The NSComboBox treats objects provided by its data source as values to be displayed in the combo box's pop-up list. If these objects aren't of common value classes—such as NSString, NSNumber, and so on—you'll need to create a custom NSFormatter to display them. See the NSFormatter class specification for more information.

When an NSComboBoxDataSource is asked to supply a data item, the NSComboBox that sends the request is provided as a parameter. This allows a single data source object to manage several sets of data, choosing the appropriate set based on the identify of the NSComboBox that sends the message.

Instance Methods

comboBox:indexOfItemWithStringValue:

– (unsigned int)**comboBox:**(NSComboBox *)*aComboBox* **indexOfItemWithStringValue:** (NSString *)*aString*

An NSComboBox uses this method to perform incremental—or "smart"—searched when the user types into the text field with the pop-up list displayed. Your implementation of this method should return the index for the item which matches *aString*, or NSNotFound if no item matches. This method is optional; if you don't provide an implementation for this method, no searches occur.

comboBox:objectValueForItemAtIndex:

– (id)**comboBox:**(NSComboBox *)*aComboBox* **objectValueForItemAtIndex:**(int)*index*

Implement this method to return the object that corresponds to the item at *index* in *aComboBox*. Your data source must implement this method.

numberOfItemsInComboBox:

– (int)**numberOfItemsInComboBox:**(NSComboBox *)*aComboBox*

Implement this method to return the number of items managed for *aComboBox* by your data source object. An NSComboBox uses this method to determine how many items it should display in its pop-up list. Your data source must implement this method.

NSDPSContextNotification

Adopted By: no OpenStep classes

Declared In: AppKit/NSDPSContext.h

Protocol Description

The NSDPSContextNotification protocol supplies information about the execution status of a sequence of PostScript commands previously sent to the Display PostScript server.

Instance Methods

contextFinishedExecuting:

– (void)**contextFinishedExecuting:**(NSDPSContext *)*context*

Notifies the receiver that the context has finished executing a batch of PostScript commands. See **notifyObjectWhenFinishedExecuting:** (NSDPSContext).

NSDraggingDestination

(informal protocol)

Category Of: NSObject

Declared In: AppKit/NSDragging.h

Description

The NSDraggingDestination informal protocol declares methods that the destination (or recipient) of a dragged image must implement. The destination automatically receives NSDraggingDestination messages as an image enters, moves around inside, and then exits or is released within the destination's boundaries.

In the text here and in the other dragging protocol descriptions, the term *dragging session* is the entire process during which an image is selected, dragged, released, and absorbed or rejected by the destination. A *dragging operation* is the action that the destination takes in absorbing the image when it's released. The *dragging source* is the object that "owns" the image that's being dragged. It's specified as an argument to the **dragImage:at:offset:event:pasteboard:source:slideBack:** message, sent to a window or view object, that instigated the dragging session.

The Dragged Image

The image that's dragged in an image-dragging session is simply an image that represents data that resides on the pasteboard. Although a dragging destination can access the image (through the **draggedImage** method described in the NSDraggingInfo protocol), its primary concern is with the pasteboard data that the image represents—the dragging operation that a destination ultimately performs is on the pasteboard data, not on the image itself.

Valid Destinations

Dragging is a visual phenomenon. To be an image-dragging destination, an object must represent a portion of screen real estate; thus, only window and view objects can be destinations. Furthermore, you must register the pasteboard types that the object will accept by sending the object a **registerForDraggedTypes:** message, defined in both NSWindow and NSView. During a dragging session, a candidate destination only receives NSDraggingDestination messages if the destination is registered for a pasteboard type that matches the type of the pasteboard data being dragged. See the NSPasteboard class specification for more information about pasteboard types.

Although NSDraggingDestination is declared as an informal protocol, the NSWindow and NSView subclasses that you create to adopt the protocol need only implement those methods that are pertinent. (The NSWindow and NSView classes provide private implementations for all of the methods.) Either a window

object or its delegate may implement these methods; however, the delegate's implementation takes precedence if there are implementations in both places.

The Sender of Destination Messages

Each of the NSDraggingDestination methods sports a single argument: *sender*, the object that invoked the method. Within its implementations of the NSDraggingDestination methods, the destination can send NSDraggingInfo protocol messages to *sender* to get more information on the current dragging session.

The Order of Destination Messages

The six NSDraggingDestination methods are invoked in a distinct order:

- As the image is dragged into the destination's boundaries, the destination is sent a **draggingEntered:** message
- While the image remains within the destination, a series of **draggingUpdated:** messages are sent.
- If the image is dragged out of the destination, **draggingExited:** is sent and the sequence of NSDraggingDestination messages stops. If it re-enters, the sequence begins again (with a new **draggingEntered:** message).
- When the image is released, it either slides back to its source (and breaks the sequence) or a **prepareForDragOperation:** message is sent to the destination, depending on the value returned by the most recent invocation of **draggingEntered:** or **draggingUpdated:**.
- If the **prepareForDragOperation:** message returned YES, a **performDragOperation:** message is sent.
- Finally, if **performDragOperation:** returned YES, **concludeDragOperation:** is sent.

Method Types

Before the image is released

– draggingEntered:
– draggingUpdated:
– draggingExited:

After the image is released

– prepareForDragOperation:
– performDragOperation:
– concludeDragOperation:

nstance Methods

concludeDragOperation:

– (void)**concludeDragOperation:**(id <NSDraggingInfo>)*sender*

Invoked when the dragging operation is complete and the previous **performDragOperation:** returned YES. The destination implements this method to perform any tidying up that it needs to do, such as updating its visual representation now that it has incorporated the dragged data. This is the last message that's sent from *sender* to the destination during a dragging session.

draggingEntered:

– (unsigned int)**draggingEntered:**(id <NSDraggingInfo>)*sender*

Invoked when a dragged image enters the destination. Specifically, this method is invoked when the mouse pointer enters the destination's bounds rectangle (if it's a view object) or its frame rectangle (if it's a window object).

This method must return a value that indicates which dragging operation the destination will perform when the image is released. In deciding which dragging operation to return, the method should evaluate the overlap between both the dragging operations allowed by the source (accessible through the **draggingSourceOperationMask** method) and the dragging operations and pasteboard data types that the destination itself supports. The returned value should be exactly one of the following:

Option	Meaning
NSDragOperationCopy	The data represented by the image will be copied.
NSDragOperationLink	The data will be shared.
NSDragOperationGeneric	The operation will be defined by the destination.
NSDragOperationPrivate	The operation is negotiated privately between the source and the destination.
NSDragOperationAll	Combines all the above.

If none of the operations is appropriate, this method should return NSDragOperationNone (this is the default response if the method isn't implemented by the destination).

The code below is a simple example of a method that responds distinctly when one of two different types of data is dragged into the destination view or window. If the dragged data is a color and the source object permits copying, the return value indicates that the destination will permit copying of the color data on the pasteboard. If the dragged data is an RTF file and the source object permits linking, the return value

indicates that the destination will permit linking of the RTF file on the pasteboard. Otherwise the code returns NSDragOperationNone, indicating that the destination will not permit any dragging operations with the data on pasteboard.

```
- (unsigned int)draggingEntered:(id <NSDraggingInfo>)sender
{
    NSPasteboard *pboard;
    NSDragOperation sourceDragMask;

    sourceDragMask = [sender draggingSourceOperationMask];
    pboard = [sender draggingPasteboard];

    if ([[pboard types] indexOfObject:NSColorPboardType] != NSNotFound) {
        if (sourceDragMask & NSDragOperationCopy) {
            return NSDragOperationCopy;
        }
    }
    if ([[pboard types] indexOfObject:NSRTFPboardType] != NSNotFound) {
        if (sourceDragMask & NSDragOperationLink) {
            return NSDragOperationLink;
        }
    }
    return NSDragOperationNone;
}
```

See also: – **draggingUpdated:**, – **draggingExited:**, – **prepareForDragOperation:**

draggingExited:

– (void)**draggingExited:**(id <NSDraggingInfo>)*sender*

Invoked when the dragged image exits the destination's bounds rectangle (in the case of a view object) or its frame rectangle (in the case of a window object).

draggingUpdated:

– (unsigned int)**draggingUpdated:**(id <NSDraggingInfo>)*sender*

Invoked periodically as the image is held within the destination. The messages continue until the image is either released or dragged out of the window or view. The return value should be one of the dragging operation options listed under the **draggingEntered:** method. The default return value (if this method isn't implemented by the destination) is the value returned by the previous **draggingEntered:** message.

This method provides the destination with an opportunity to modify the dragging operation depending on the position of the mouse pointer inside of the destination view or window object. For example, you may have several graphics or areas of text contained within the same view and wish to tailor the dragging

operation, or to ignore the drag event completely, depending upon which object is underneath the mouse pointer at the time when the user releases the dragged image and the **performDragOperation:** method is invoked.

You typically examine the contents of the pasteboard in the **draggingEntered:** method, where this examination is performed only once, rather than in the **draggingUpdated:** method, which is invoked multiple times.

Only one destination at a time receives a sequence of **draggingUpdated:** messages. If the mouse pointer is within the bounds of two overlapping views that are both valid destinations, the uppermost view receives these messages until the image is either released or dragged out.

See also: – **draggingExited:**, – **prepareForDragOperation:**

performDragOperation:

– (BOOL)**performDragOperation:**(id <NSDraggingInfo>)*sender*

Invoked after the released image has been removed from the screen and the previous **prepareForDragOperation:** message has returned YES. The destination should implement this method to do the real work of importing the pasteboard data represented by the image. If the destination accepts the data, it returns YES, otherwise it returns NO. The default is to return NO.

See also: – **concludeDragOperation:**

prepareForDragOperation:

– (BOOL)**prepareForDragOperation:**(id <NSDraggingInfo>)*sender*

Invoked when the image is released, if the most recent **draggingEntered:** or **draggingUpdated:** message returned an acceptable drag-operation value. Returns YES if the receiver agrees to perform the drag operation and NO if not.

See also: – **performDragOperation:**

NSDraggingInfo

Adopted By: no Rhapsody classes

Declared In: AppKit/NSDragging.h

Protocol Description

The NSDraggingInfo protocol declares methods that supply information about a *dragging session* (see the NSDraggingDestination protocol for definitions of dragging terms). NSDraggingInfo methods are designed to be invoked from within a class's implementation of NSDraggingDestination informal protocol methods. The Application Kit automatically passes an object that conforms to the NSDraggingInfo protocol as the argument to each of the methods defined by NSDraggingDestination. NSDraggingInfo messages should be sent to this object; you never need to create a class that implements the NSDraggingInfo protocol.

Method Types

Dragging-session information

– draggingSource
– draggingSourceOperationMask
– draggingDestinationWindow
– draggingPasteboard
– draggingSequenceNumber
– draggingLocation

Image information

– draggedImage
– draggedImageLocation

Sliding the image

– slideDraggedImageTo:

nstance Methods

draggedImage

– (NSImage *)**draggedImage**

Returns the image being dragged. This image object visually represents the data put on the pasteboard during the drag operation; however, it is the pasteboard data and not this image that are ultimately utilized in the dragging operation.

See also: – **draggedImageLocation**

draggedImageLocation

– (NSPoint)**draggedImageLocation**

Returns the current location of the dragged image's origin in the base coordinate system of the destination object's window. The image moves along with the mouse pointer (the position of which is given by **draggingLocation**) but may be positioned at some offset.

See also: – **draggedImage**

draggingDestinationWindow

– (NSWindow *)**draggingDestinationWindow**

Returns the destination window for the dragging operation. Either this window is the destination itself, or it contains the view object that is the destination.

draggingLocation

– (NSPoint)**draggingLocation**

Returns the current location of the mouse pointer in the base coordinate system of the destination object's window.

See also: – **draggedImageLocation**

draggingPasteboard

– (NSPasteboard *)**draggingPasteboard**

Returns the pasteboard object that holds the data being dragged. The dragging operation that is ultimately performed utilizes this pasteboard data and not the image returned by the **draggedImage** method.

draggingSequenceNumber

– (int)**draggingSequenceNumber**

Returns a number that uniquely identifies the dragging session.

draggingSource

– (id)**draggingSource**

Returns the source, or owner, of the dragged data or **nil** if the source isn't in the same application as the destination. The dragging source implements methods from the NSDraggingSource informal protocol.

draggingSourceOperationMask

– (unsigned int)**draggingSourceOperationMask**

Returns the dragging operation mask declared by the dragging source (through its **draggingSourceOperationMaskForLocal:** method). If the source permits dragging operations, the elements in the mask will be one or more of the following, combined using the C bitwise OR operator:

Option	Meaning
NSDragOperationCopy	The data represented by the image can be copied.
NSDragOperationLink	The data can be shared.
NSDragOperationGeneric	The operation can be defined by the destination.
NSDragOperationPrivate	The operation is negotiated privately between the source and the destination.
NSDragOperationAll	Combines all the above.

If the source does not permit any dragging operations, then method should return NSDragOperationNone.

If the user is holding down a modifier key during the dragging session and the source doesn't prohibit modifier keys from affecting the drag operation (through its **ignoreModifierKeysWhileDragging** method), then the operating system combines the dragging operation value that corresponds to the modifier key (see the descriptions below) with the source's mask using the C bitwise AND operator.

On Mach the modifier keys are associated with the dragging operation options shown below,

Modifier Key	Dragging Option
Control	NSDragOperationLink
Alternate	NSDragOperationCopy
Command	NSDragOperationGeneric

while on Windows the modifier keys are associated with the following dragging operation options.

Modifier Key	Dragging Option
Control	NSDragOperationCopy
Shift-Control	NSDragOperationLink
Alternate	NSDragOperationCopy

slideDraggedImageTo:

– (void)**slideDraggedImageTo:**(NSPoint)*aPoint*

Slides the image to *aPoint*, a specified location in the screen coordinate system. This method can be used to snap the image down to a particular location. It should only be invoked from within the destination's implementation of **prepareForDragOperation:**—in other words, after the user has released the image but before it's removed from the screen.

NSDraggingSource

(informal protocol)

Category Of: NSObject

Declared In: AppKit/NSDragging.h

Description

The NSDraggingSource informal protocol declares methods that are implemented by the source object in a dragging session (see the NSDraggingDestination protocol for definitions of dragging terms). The *dragging source* is specified as an argument to the **dragImage:at:offset:event:pasteboard:source: slideBack:** message, sent to a window or view object to initiate the dragging session.

Of the methods declared below, only **draggingSourceOperationMaskForLocal:** must be implemented. The other methods are invoked only if the dragging source implements them. All four methods are invoked automatically during a dragging session—you never send an NSDraggingSource message directly to an object.

Method Types

Specifying dragging options

– draggingSourceOperationMaskForLocal:
– ignoreModifierKeysWhileDragging

Responding to dragging sessions

– draggedImage:beganAt:
– draggedImage:endedAt:deposited:

Instance Methods

draggedImage:beganAt:

– (void)**draggedImage:**(NSImage *)*anImage* **beganAt:**(NSPoint)*aPoint*

Invoked when *anImage* is displayed but before it starts following the mouse. *aPoint* is the origin of the image in screen coordinates. This method provides the source object with an opportunity to respond to the initiation of a dragging session. For example, you might choose to have the source give a visual indication to the user that data is being dragged from the source.

See also: **– convertScreenToBase:** (NSWindow), **– convertBaseToScreen:** (NSWindow), **– convertPoint:fromView:** (NSView), **– convertPoint:toView:** (NSView)

draggedImage:endedAt:deposited:

– (void)**draggedImage:**(NSImage *)*anImage*
endedAt:(NSPoint)*aPoint*
deposited:(BOOL)*flag*

Invoked after *anImage* has been released and the dragging destination has been given a chance to operate on the data it represents. *aPoint* is the location of the image's origin in the screen coordinate system when it was released. A YES value for *flag* indicates that the destination accepted the dragged data, while a NO value indicates that it was rejected.

This method provides the source object with an opportunity to respond to either a successful or a failed dragging session. For example, if you are moving data from one location to another, you could use this method to make the source data disappear from its previous location, if the dragging session is successful, or reset itself to its previous state, in the event of a failure.

See also: **– convertScreenToBase:** (NSWindow), **– convertBaseToScreen:** (NSWindow), **– convertPoint:fromView:** (NSView), **– convertPoint:toView:** (NSView)

draggingSourceOperationMaskForLocal:

– (unsigned int)**draggingSourceOperationMaskForLocal:**(BOOL)*flag*

This is the only NSDraggingSource method that must be implemented by the source object. It should return a mask, built by combining the applicable constants listed below using the C bitwise OR operator. You should use this mask to indicate which types of dragging operations the source object will allow to be performed on the dragged image's data. A YES value for *flag* indicates that the candidate destination object (the window or view over which the dragged image is currently poised) is in the same application as the source, while a NO value indicates that the destination object is in a different application.

Option	Meaning
NSDragOperationCopy	The data represented by the image can be copied.
NSDragOperationLink	The data can be shared.
NSDragOperationGeneric	The operation can be defined by the destination.
NSDragOperationPrivate	The operation is negotiated privately between the source and the destination.
NSDragOperationAll	Combines all the above.

If the source does not permit any dragging operations, then it should return NSDragOperationNone.

ignoreModifierKeysWhileDragging

– (BOOL)**ignoreModifierKeysWhileDragging**

Sets whether the use of the modifier keys should have no effect on the type of operation performed. If this method is not implemented or returns NO, then the user can tailor the drag operation by holding down a modifier key during the drag. The dragging option that corresponds to the modifier key is combined with the source's mask (as set with the **draggingSourceOperationMaskForLocal:** method) using the C bitwise AND operator. See the description for the **draggingSourceOperationMask** method in the NSDraggingInfo protocol specification for more information about dragging masks and modifier keys.

NSIgnoreMisspelledWords

Adopted By: NSText

Declared In: AppKit/NSSpellProtocol.h

Protocol Description

Implement this protocol to have the Ignore button in the Spelling panel function properly. The Ignore button allows the user to accept a word that the spelling checker believes is misspelled. In order for this action to update the “ignored words” list for the document being checked, the NSIgnoreMisspelledWords protocol must be implemented.

This protocol is necessary because a list of ignored words is useful only if it pertains to the entire document being checked, but the spelling checker (NSSpellChecker object) does not check the entire document for spelling at once. The spelling checker returns as soon as it finds a misspelled word. Thus, it checks only a subset of the document at any one time. The user usually wants to check the entire document, so usually several spelling checks are run in succession until no misspelled words are found. This protocol allows the list of ignored words to be maintained per-document, even though the spelling checks are not run per-document.

The NSIgnoreMisspelledWords protocol specifies a method, **ignoreSpelling:**, which should be implemented like this:

```
- (void)ignoreSpelling:(id)sender
{
    [[NSSpellChecker sharedSpellChecker] ignoreWord:[[sender selectedCell]
        stringValue] inSpellDocumentWithTag:myDocumentTag];
}
```

The second argument to the NSSpellChecker method **ignoreWord:inSpellDocumentWithTag:** is a tag that the NSSpellChecker can use to distinguish the documents being checked. (See the discussion of “Matching a List of Ignored Words With the Document It Belongs To” in the description of the NSSpellChecker class.) Once the NSSpellChecker has a way to distinguish the various documents, it can append new ignored words to the appropriate list.

To make the ignored words feature useful, the application must store a document’s ignored words list with the document. See the NSSpellChecker class description for more information.

nstance Methods

ignoreSpelling:

– (void)**ignoreSpelling:**(id)*sender*

Implement to allow an application to ignore misspelled words on a document-by-document basis. This message is sent by the NSSpellChecker instance to the object whose text is being checked.

Implement this method by using the code shown in the protocol description.

NSMenuItem

Adopted By: NSMenuItem

Conforms To: NSCoding,
NSCopying
NSObject (NSObject)

Declared In: AppKit/NSMenuItem.h

Warning: The NSMenuItem protocol will be removed from the Application Kit in the Premier release of Rhapsody. The NSMenuItem class will solely assume all associated functionality. This change does not affect binary compatibility between different versions of projects, but might cause failures in project builds. To adapt your projects to this change, alter all references to the protocol (for example, “id <NSMenuItem>”) to references to the class (“NSMenuItem”).

Protocol Description

The NSMenuItem protocol declares methods that are used to manipulate command items in menus. The NSMenuItem class adopts this protocol, implementing all methods the protocol declares, and provides the basic functionality of command items. With some implementations of the OpenStep specification (including OPENSTEP), you cannot replace the NSMenuItem class with a different class which conforms to the NSMenuItem protocol. You may, however, subclass the NSMenuItem class if necessary.

The methods declared by the NSMenuItem protocol allow you to set the titles, actions, targets, tags, images, enabled states, and similar attributes of individual menu items, as well as to obtain the current values of these attributes. As implemented for the NSMenuItem class, a menu item, whenever one of its attributes changes, notifies the associated NSMenu via the **itemChanged:** method. The protocol also allows a conforming object to set keyboard equivalents and (for Microsoft Windows) mnemonics for menu items. See the sections below for more on this functionality.

See the NSMenu, NSMenuView, and NSMenuItemCell class specifications and the NSMenuValidation protocol specification for more information on menus.

Keyboard Equivalents

An object conforming to the NSMenuItem protocol can be assigned a keyboard eqivalent, so that when the user types a character the menu item’s action is sent. The keyboard eqivalent is defined in two parts. First is the basic key equivalent, which must be a Unicode character that can be generated by a single key press without modifier keys (Shift excepted). It is also possible to use a sequence of Unicode characters so long

as the user's key mapping is able to generate the sequence with a single key press. The basic key equivalent is set using **setKeyEquivalent:** and returned by **keyEquivalent**. The second part defines the modifier keys that must also be pressed. This is set using **setKeyEquivalentModifierMask:** and returned by **keyEquivalentModifierMask**. The modifier mask by default includes NSCommandKeyMask, and may also include the masks for the Shift, Alternate, or other modifier keys. Specifying keyboard equivalents in two parts allows you to define a modified keyboard equivalent without having to know which character is generated by the basic key plus the modifier. For example, you can define the keyboard equivalent Command-Alt-f without having to know which character is generated by typing Alt-f.

Certain methods in the NSMenuItem protocol can override assigned keyboard equivalents with those the user has specified in the defaults system. The **setUsesUserKeyEquivalents:** protocol method turns this behavior on or off, and **usesUserKeyEquivalents** returns its status. To determine the user-defined key equivalent for an NSMenuItem object, invoke the **userKeyEquivalent** instance method. If user-defined key equivalents are active and an NSMenuItem object has a user-defined key equivalent, its **keyEquivalent** method returns the user-defined key equivalent and not the one set using **setKeyEquivalent:**.

Mnemonics

On certain platforms, currently including Microsoft Windows, an object conforming to the NSMenuItem protocol can also be assigned a mnemonic. Mnemonics can be assigned on other platforms as well, however, they won't have any effect. Mnemonics are represented by an underlined character in the title of a menu item. The mnemonic can be any character that can be generated by a single key press without modifier keys (Shift excepted). When the menu is active, the user can type the underlined character in the menu item in order to activate that menu item. On Microsoft Windows a user activates the menu by pressing the Alternate key. A particular mnemonic character should only be used once within the set of menu items contained either in the same menu as the menu item or in the application's main menu.

Radio-Style Grouping

By using a few methods of the NSMenuItem protocol, you can implement radio-style groupings of menu commands. In other words, you can have a grouping of menu commands (usually segregated visually with separator items) and only one command in the group can be selected; the selected item is marked by an image, usually a radio-button image, but sometimes a checkmark. If the user selects another command in the group, the previous command is unmarked and the selected command displays the image. As an example of a radio-style grouping, a game could have three commands to indicate the level of play: Beginner, Intermediate, and Advanced.

To implement this feature, first set the images you want to use for the possible command states: "on," "off," and "mixed" (the last is useful for triple-state or indeterminate situations). To set the image, use the commands **setOnStateImage:**, **setOffStateImage:**, and **setMixedStateImage:**. The default image for the "on" state is a checkmark (NSMenuCheckmark) and for the "mixed" state the image is a dash (NSMenuMixedState). The "off" state typically has no image. The radio-button image (which you must set explicitly) is NSMenuRadio.

In an action method that responds to all commands in the group use **setState:** to uncheck the menu item that is currently marked:

```
[curItem setState:NSOffState];
```

Then mark the newly selected command:

```
[sender setState:NSOnState];
```

Method Types

Creating conforming NSMenuItem objects
– initWithTitle:action:keyEquivalent:

Enabling a menu item
– setEnabled:
– isEnabled

Setting the target and action
– setTarget:
– target
– setAction:
– action

Setting the title
– setTitle:
– title

Setting the tag
– setTag:
– tag

Setting the state
– setState:
– state

Setting the image
– setImage:
– image
– setOnStateImage:
– onStateImage
– setOffStateImage:
– offStateImage
– setMixedStateImage:
– mixedStateImage

Managing submenus

– setSubmenu:
– submenu
– hasSubmenu

Getting a separator item

+ separatorItem
– isSeparatorItem

Setting the owning menu

– setMenu:
– menu

Managing key equivalents

– setKeyEquivalent:
– keyEquivalent
– setKeyEquivalentModifierMask:
– keyEquivalentModifierMask

Managing mnemonics

– setMnemonicLocation:
– mnemonicLocation
– setTitleWithMnemonic:
– mnemonic

Managing user key equivalents

+ setUsesUserKeyEquivalents:
+ usesUserKeyEquivalents
– userKeyEquivalent

Representing an object

– setRepresentedObject:
– representedObject

Class Methods

separatorItem

+ (id <NSMenuItem>)**separatorItem**

Returns a menu item that is used to separate logical groups of menu commands. This menu item is disabled. The default separator item is a simple horizontal line.

See also: – **isSeparatorItem**, – **setEnabled:**

setUsesUserKeyEquivalents:

+ (void)**setUsesUserKeyEquivalents:**(BOOL)*flag*

If *flag* is YES, menu items conform to user preferences for key equivalents; otherwise, the key equivalents originally assigned to the menu items are used.

See also: **+ usesUserKeyEquivalents, – userKeyEquivalent**

usesUserKeyEquivalents

+ (BOOL)**usesUserKeyEquivalents**

Returns YES if menu items conform to user preferences for key equivalents; otherwise, returns NO.

See also: **+ setUsesUserKeyEquivalents:, – userKeyEquivalent**

nstance Methods

action

– (SEL)**action**

Returns the receiver's action method.

See also: **– target, – setAction:**

hasSubmenu

– (BOOL)**hasSubmenu**

Returns YES if the receiver has a submenu, NO if it doesn't.

See also: **– setSubmenu:forItem:**(NSMenu)

image

– (NSImage *)**image**

Returns the image displayed by the receiver, or **nil** if it displays no image.

See also: **– setImage:**

initWithTitle:action:keyEquivalent:

– (id)**initWithTitle:**(NSString *)*itemName*
action:(SEL)*anAction*
keyEquivalent:(NSString *)*charCode*

Returns an initialized instance of an object that conforms to the NSMenuItem protocol. The arguments *itemName* and *charCode* must not be **nil** (if there is no title or key equivalent, specify an empty NSString). The *anAction* argument must be a valid selector or NULL For instances of the NSMenItem class, the default initial state is NSStateOff, the default on-state image is a checkmark, and the default mixed-state image is a dash.

isEnabled

– (BOOL)**isEnabled**

Returns YES if the receiver is enabled, NO if not.

See also: – **setEnabled:**

isSeparatorItem

– (BOOL)**isSeparatorItem**

Returns whether the receiver is a separator item (that is, a menu item used to visually segregate related menu items).

See also: + **separatorItem**

keyEquivalent

– (NSString *)**keyEquivalent**

Returns the receiver's unmodified keyboard equivalent, or the empty string if one hasn't been defined. Use **keyEquivalentModifierMask** to determine the modifier mask for the key equivalent.

See also: **– userKeyEquivalent, – mnemonic, – setKeyEquivalent:**

keyEquivalentModifierMask

– (unsigned int)**keyEquivalentModifierMask**

Returns the receiver's keyboard equivalent modifier mask.

See also: **– setKeyEquivalentModifierMask:**

menu

– (NSMenu *)**menu**

Returns the menu to which the receiver belongs, or **nil** if no menu has been set.

See also: – **setMenu:**

mixedStateImage

– (NSImage *)**mixedStateImage**

Returns the image used to depict a “mixed state.” A mixed state is useful for indicating “off” and “on” attribute values in a group of selected objects, such as a selection of text containing bold and plain (non-bolded) worlds.

See also: – **setMixedStateImage:**

mnemonic

– (NSString *)mnemonic

Returns the character in the menu item title that appears underlined for use as a mnemonic. If there is no mnemonic character, returns an empty string.

See also: – **setTitleWithMnemonic:**

mnemonicLocation

– (unsigned int)**mnemonicLocation**

Returns the position of the underlined character in the menu item title used as a mnemonic. The position is the zero based index of that character in the title string. If the receiver has no mnemonic character, returns NSNotFound.

See also: – **setMnemonicLocation:**

offStateImage

– (NSImage *)offStateImage

Returns the image used to depict the receiver’s “off” state, or **nil** if the image has not been set.

See also: – **setOffStateImage:**

onStateImage

– (NSImage *)**onStateImage**

Returns the image used to depict the receiver's "on" state, or **nil** if the image has not been set.

See also: – **setOnStateImage:**

representedObject

– (id)**representedObject**

Returns the object that the receiving menu item represents. For example, you might have a menu list the names of views that are swapped into the same panel. The represented objects would be the appropriate NSView objects. The user would then be able to swtich back and forth between the different views that are displayed by selecting the various menu items.

See also: – **tag**, – **setRepresentedObject:**

setAction:

– (void)**setAction:**(SEL)*aSelector*

Sets the receiver's action method to *aSelector*.

See also: – **setTarget:**, – **action**

setEnabled:

– (void)**setEnabled:**(BOOL)*flag*

Sets whether the receiver is enabled based on *flag*. If a menu item is disabled, it's keyboard equivalent and mnemonic are also disabled. See the NSMenuValidation informal protocol specification for cautions regarding this method.

See also: – **isEnabled**

setImage:

– (void)**setImage:**(NSImage *)*menuImage*

Set's the receiver's image to *menuImage*. If *menuImage* is **nil**, the current image (if any) is removed. This image is not affected by changes in menu-item state.

See also: – **image**

setKeyEquivalent:

– (void)**setKeyEquivalent:**(NSString *)*aString*

Sets the receiver's unmodified key equivalent to *aString*. If you want to remove the key equivalent from a menu item, pass an empty string (@"") for *aString* (never pass **nil**). Use **setKeyEquivalentModifierMask:** to set the appropriate mask for the modifier keys for the key equivalent.

See also: **– setMnemonicLocation:, – keyEquivalent**

setKeyEquivalentModifierMask:

– (void)**setKeyEquivalentModifierMask:**(unsigned int)*mask*

Sets the receiver's keyboard equivalent modifiers (indicating modifiers such as the Shift or Alternate keys) to those in *mask*. *mask* is an integer bit field containing any of these modifier key masks, combined using the C bitwise OR operator:

NSShiftKeyMask
NSAlternateKeyMask
NSCommandKeyMask

On Mach, you should always set NSCommandKeyMask in *mask*; on Microsoft Windows, this is not required.

NSShiftKeyMask is relevant only for function keys; that is, for key events whose modifier flags include NSFunctionKeyMask. For all other key events NSShiftKeyMask is ignored and characters typed while the Shift key is pressed are interpreted as the shifted versions of those characters; for example, Command-Shift-'c' is interpreted as Command-'C'.

See the NSEvent class specification for more information about modifier mask values.

See also: **– keyEquivalentModifierMask**

setMenu:

– (void)**setMenu:**(NSMenu *)*aMenu*

Sets the receiver's menu to *aMenu*. This method is invoked by the owning NSMenu when the receiver is added or removed. You shouldn't have to invoke this method in your own code, although it can be overriden to provide specialized behavior.

See also: – **menu**

setMixedStateImage:

– (void)**setMixedStateImage:**(NSImage *)*itemImage*

Sets the image of the receiver that indicates a”mixed” state, that is, a state neither “on” or “off.” If *itemImage* is **nil**, any current mixed-state image is removed.

See also: – **mixedStateImage**, – **setOffStateImage:**, – **setOnStateImage:**, – **setState:**

setMnemonicLocation:

– (void)**setMnemonicLocation:**(unsigned int)*location*

Sets the character of the menu item title at *location* that is to be underlined. *location* must be between 0 and 254. This character identifies the access key on Windows by which users can access the menu item.

See also: – **mnemonicLocation**

setOffStateImage:

– (void)**setOffStateImage:**(NSImage *)*itemImage*

Sets the image of the receiver that indicates an “off” state. If *itemImage* is **nil**, any current off-state image is removed.

See also: – **offStateImage**, – **setMixedStateImage:**, – **setOnStateImage:**, – **setState:**

setOnStateImage:

– (void)**setOnStateImage:**(NSImage *)*itemImage*

Sets the image of the receiver that indicates an “on” state. If *itemImage* is **nil**, any current off-state image is removed.

See also: – **onStateImage**, – **setMixedStateImage:**, – **setOffStateImage:**, – **setState:**

setRepresentedObject:

– (void)**setRepresentedObject:**(id)*anObject*

Sets the object represented by the receiver to *anObject*. By setting a represented object for a menu item you make an association between the menu item and that object. The represented object functions as a more specific form of tag that allows you to associate any object, not just an **int**, with the items in a menu.

For example, an NSView object might be associated with a menu item—when the user chooses the menu item, the represented object is fetched and displayed in a panel. Several menu items might control the display of multiple views in the same panel.

See also: – **setTag:**, – **representedObject**

setState:

– (void)**setState:**(int)*itemState*

Sets the state of the receiver to *itemState*, which should be one of NSOffState, NSOnState, or NSMixedState. The image associated with the new state is displayed to the left of the menu item.

See also: – **state**, – **setMixedStateImage:**, – **setOffStateImage:**, – **setOnStateImage:**

setSubmenu:

– (void)**setSubmenu:**(NSMenu *)*aSubmenu*

Sets the submenu of the receiver to *aSubmenu*. The default implementation of the NSMenuItem class raises an exception if *aSubmenu* already has a supermenu.

See also: – **submenu**, – **hasSubmenu**

setTag:

– (void)**setTag:**(int)*anInt*

Sets the receiver's tag to *anInt*.

See also: – **setRepresentedObject:**, – **tag**

setTarget:

– (void)**setTarget:**(id)*anObject*

Sets the receiver's target to *anObject*.

See also: **– setAction:, – target**

setTitle:

– (void)**setTitle:**(NSString *)*aString*

Sets the receiver's title to *aString*.

See also: – **title**

setTitleWithMnemonic:

– (void)**setTitleWithMnemonic:**(NSString *)*aString*

Sets the title of a menu item with a character underlined to denote an access key (Windows only). Use an ampersand character to mark the character (the one following the ampersand) to be underlined. For example, the following message causes the 'c' in 'Receive' to be underlined:

```
[aMenuItem setTitleWithMnemonic:NSLocalizedString(@"Re&ceive")];
```

See also: – **mnemonic**, – **setMnemonicLocation:**

state

– (int)**state**

Returns the state of the receiver, which is NSOffState (the default), NSOnState, or NSMixedState.

See also: – **setState:**

submenu

– (NSMenu *)**submenu**

Returns the submenu associated with the receiving menu item, or **nil** if no submenu is associated with it. In the implementation of the NSMenuItem class, if the receiver responds YES to **hasSubmenu**, the submenu is returned.

See also: – **hasSubmenu**, – **setSubmenu**:

tag

– (int)**tag**

Returns the receiver's tag.

See also: – **representedObject**, – **setTag:**

target

– (id)**target**

Returns the receiver's target.

See also: **– action, – setTarget:**

title

– (NSString *)**title**

Returns the receiver's title.

See also: **– setTitle:**

userKeyEquivalent

– (NSString *)**userKeyEquivalent**

Returns the user-assigned key equivalent for the receiver.

See also: **– keyEquivalent**

NSMenuValidation

(informal protocol)

Category Of: NSObject

Declared In: AppKit/NSMenu.h

Protocol Description

This informal protocol allows your application to update the enabled or disabled status of an NSMenuItem. It declares only one method, **validateMenuItem:**. By default, every time a user event occurs, NSMenu automatically enables and disables each visible menu item based on criteria described in "Automatic Updating of NSMenuItems," below. Implement **validateMenuItem:** in cases where you want to override NSMenu's default enabling scheme.

NSMenuItems can be enabled or disabled in two ways: explicitly, by sending the **setEnabled:** message, or automatically, as described below. Automatic updating can be turned on and off with NSMenu's **setAutoenablesItems:** message.

Automatic Updating of NSMenuItems

Whenever a user event occurs, the NSMenu object updates the status of every one of its visible menu items. To update the status of a menu item, an NSMenu tries to find the object that responds to the NSMenuItem's action message. It searches the following objects in the given order until it finds one that responds to the action message (note that it doesn't actually send the action message):

- The NSMenuItem's target. If the target is non-**nil**, the search ends here whether the target responds or not.
- The key window's responder chain, starting with its first responder.
- The key window itself.
- The key window's delegate.
- The main window's responder chain, starting with its first responder.
- The main window itself.
- The main window's delegate.
- The NSApplication object.
- The NSApplication object's delegate.

If none of these objects responds to the action message, the menu item is disabled. If NSMenu finds an object that responds to the action message, it then checks to see if that object responds to the **validateMenuItem:** method (the method declared in this informal protocol). If **validateMenuItem:** is not

implemented in that object, the menu item is enabled. If it is implemented, the return value of **validateMenuItem:** indicates whether the menu item should be enabled or disabled.

Here is an example of using **validateMenuItem:** to override automatic enabling. If your application has a Copy menu item that sends the **copy:** action message to the first responder, that menu item is automatically enabled any time an object that responds to **copy:**, such as an NSText object, is the first responder of the key or main window. If you create a class whose instances might become the first responder, and which doesn't support copying of everything it allows the user to select, you should implement **validateMenuItem:** in that class. **validateMenuItem:** will then return NO if items that can't be copied are selected (or if no items are selected) and YES if all items in the selection can be copied. By implementing **validateMenuItem:**, you can have the Copy menu item disabled even though the target object does implement the **copy:** method. If a class never permits copying, then you simply omit an implemention of **copy:** in that class, and the Copy menu item is disabled automatically whenever an instance of that class is the first responder.

If you send a **setEnabled:** message to enable or disable a menu item when automatic updating is turned on (with NSMenu's **setAutoEnablesItems:**), other objects might undo what you have done after another user event occurs. Hence you can never be sure that the menu item will remain the way you set it. If your application must use **setEnabled:**, turn off the automatic enabling of menu items in order to get predictable results.

nstance Methods

validateMenuItem:

– (BOOL)**validateMenuItem:**(NSMenuItem *)*aMenuItem*

Implemented to override the default action of enabling or disabling *aMenuItem*. The object implementing this method must be the target of *aMenuItem*. It returns YES to enable the *aMenuItem*, NO to disable it. You can determine which menu item *aMenuItem* is by querying it for its title, tag, or action.

The following example beeps and disables the menu item "Next Record" if the selected line in a table view is the last one; conversely, it beeps and disables the menu item "Prior Record" if the selected row is the first one in the table view. (**countryKeys** is an array of names appearing in the table view.)

```
- (BOOL)validateMenuItem:(NSMenuItem *)anItem
{
    int row = [tableView selectedRow];
    if ([[anItem title] isEqualToString:@"Next Record"] &&
        (row == [countryKeys indexOfObject:[countryKeys lastObject]])) {
        return NO;
    }
    if ([[anItem title] isEqualToString:@"Prior Record"] && row == 0 ) {
        return NO;
    }
    return YES;
}
```

NSNibAwaking

(informal protocol)

Category Of: NSObject

Declared In: AppKit/NSNibLoading.h

Protocol Description

This informal protocol consists of a single method, **awakeFromNib**. Classes can implement this method to perform final initialization of state after objects have been loaded from an Interface Builder archive.

Instance Methods

awakeFromNib

– (void)**awakeFromNib**

Implemented to prepare the receiver for service after it has been loaded from an Interface Builder archive, or *nib file*. An **awakeFromNib** message is sent to each object loaded from the archive, but only if it can respond to the message, and only after all the objects in the archive have been loaded and initialized. When an object receives an **awakeFromNib** message, it's guaranteed to have all its outlet instance variables set.

Note: This method is also sent during Interface Builder's test mode to objects instantiated from loaded palettes, which include executable code for the objects. It isn't sent to objects defined solely by using the Classes display of the nib file window in Interface Builder.

When an Interface Builder archive is loaded into an application, each custom object from the archive is first initialized with an **init** message, or **initWithFrame:** if the object is a kind of NSView. It's then more specifically initialized with the properties that it was configured with using Interface Builder. This part of the initialization process uses any **set***Variable***:** methods that are available (where *variable* is the name of an instance variable whose value was set in Interface Builder). Finally, after all the objects are fully initialized, each receives an **awakeFromNib** message.

The order in which objects are loaded from the archive is not guaranteed. Therefore, it's possible for a **set***Variable***:** message to be sent to an object before its companion objects have been unarchived. For this reason, **set***Variable***:** methods should not send messages to other objects in the archive. However, messages to other objects can safely be sent from within **awakeFromNib**—by which time it's assured that all the objects are unarchived and initialized (though not necessarily awakened, of course).

Typically, **awakeFromNib** is implemented for classes whose instances are used as the owners of a loaded nib file (shown as "File's Owner" in Interface Builder). Such a class has the express purpose of connecting the loaded objects with objects in the application, and can thereafter be disposed of, or remain in the

capacity of a controller or coordinator for the loaded objects. For example, suppose that a nib file contains two custom views that must be positioned relative to each other at run time. Trying to position them when either one of the views is initialized (in **initWithCoder:** or a **set***Variable***:** method) might fail, since the other views might not be unarchived and initialized yet. However, it can be done in the nib file owner's **awakeFromNib** method (**firstView** and **secondView** are outlets of the file's owner):

```
- (void)awakeFromNib
{
    NSRect viewFrame;

    if ([[self superclass] instancesRespondToSelector:@selector(awakeFromNib)]) {
        [super awakeFromNib];
    }
    viewFrame = [firstView frame];
    viewFrame.origin.x += viewFrame.size.width;
    [secondView setFrame:viewFrame];
    return;
}
```

Note the testing of the superclass before invoking its implementation of **awakeFromNib**. The Application Kit declares a prototype for this method, but doesn't implement it. Because there's no default implementation of **awakeFromNib**, be sure to invoke it only when the object does in fact respond.

See also: + **loadNibNamed:owner:** (NSBundle Additions),
– **awakeAfterUsingCoder** (NSObject class of the Foundation Kit),
– **initWithCoder:** (NSCoding protocol of the Foundation Kit),
+ **initialize** (NSObject class of the Foundation Kit)

NSServicesRequests

(informal protocol)

Category Of: NSObject

Declared In: AppKit/NSApplication.h

Protocol Description

This informal protocol consists of two methods, **writeSelectionToPasteboard:types:** and **readSelectionFromPasteboard:**. The first is implemented to provide data to a remote service, and the second to receive any data the remote service might send back. Both respond to messages that are generated when the user chooses a command from the Services menu.

Instance Methods

readSelectionFromPasteboard:

- (BOOL)**readSelectionFromPasteboard:**(NSPasteboard *)*pboard*

Implemented to replace the current selection (that is, the text or objects that are currently selected) with data read from the Pasteboard object *pboard*. The data would have been placed in the pasteboard by another application in response to a remote message from the Services menu. A **readSelectionFromPasteboard:** message is sent to the same object that previously received a **writeSelectionToPasteboard:types:** message.

There's no default **readSelectionFromPasteboard:** method. The Application Kit declares a prototype for this method, but doesn't implement it.

writeSelectionToPasteboard:types:

- (BOOL)**writeSelectionToPasteboard:**(NSPasteboard *)*pboard* **types:**(NSArray *)*types*

Implemented to write the current selection to the Pasteboard object *pboard*. The selection should be written as one or more of the data types listed in *types*. After writing the data, this method should return YES. If for any reason it can't write the data, it should return NO.

A **writeSelectionToPasteboard:types:** message is sent to the first responder when the user chooses a command from the Services menu, but only if the receiver didn't return **nil** to a previous **validRequestorForSendType:returnType:** message.

After this method writes the data to the pasteboard, a remote message is sent to the application that provides the service the user requested. If the service provider supplies return data to replace the selection, the first responder will then receive a **readSelectionFromPasteboard:** message.

There's no default **writeSelectionToPasteboard:types:** method. The Application Kit declares a prototype for this method, but doesn't implement it.

See also: – **validRequestorForSendType:returnType:** (NSResponder class)

NSTableDataSource

(informal protocol)

Category Of: NSObject

Declared In: AppKit/NSTableView.h

Category Description

The NSTableDataSource category declares the methods that an NSTableView uses to access the contents of its data source object. It determines how many rows to display by sending a **numberOfRowsInTableView:** message, and accesses individual values with the **tableView: objectValueForTableColumn:row:** and **tableView:setObjectValue:forTableColumn:row:** methods. A data source must implement the first two methods to work with an NSTableView, but if it doesn't implement the third the NSTableView simply provides read-only access to its contents.

The NSTableView treats objects provided by its data source as values to be displayed in NSCell objects. If these objects aren't of common value classes—such as NSString, NSNumber, and so on—you'll need to create a custom NSFormatter to display them. See the NSFormatter class specification for more information.

Suppose that an NSTableView's column identifiers are set up as NSStrings containing the names of attributes for the column, such as "Last Name", "City", and so on, and that the data source stores its records as an NSMutableArray, called **records**, of NSMutableDictionary objects using those names as keys. Here's a small example, given as an ASCII property list:

```
(
    {
        "Last Name" = Anderson;
        "First Name" = James;
        Abode = apartment;
        City = "San Francisco";
    },
    {
        "Last Name" = Beresford;
        "First Name" = Keith;
        Abode = apartment;
        City = "Redwood City";
    }
)
```

With such a record structure, this implementation of **tableView:objectValueForTableColumn:row:** suffices to retrieve values for the NSTableView:

```
- (id)tableView:(NSTableView *)aTableView
    objectValueForTableColumn:(NSTableColumn *)aTableColumn
    row:(int)rowIndex
{
    id theRecord, theValue;

    NSParameterAssert(rowIndex >= 0 && rowIndex < [records count]);
    theRecord = [records objectAtIndex:rowIndex];
    theValue = [theRecord objectForKey:[aTableColumn identifier]];
    return theValue;
}
```

Here's the corresponding method for setting values:

```
- (void)tableView:(NSTableView *)aTableView
    setObjectValue:anObject
    forTableColumn:(NSTableColumn *)aTableColumn
    row:(int)rowIndex
{
    id theRecord;

    NSParameterAssert(rowIndex >= 0 && rowIndex < [records count]);
    theRecord = [records objectAtIndex:rowIndex];
    [theRecord setObject:anObject forKey:[aTableColumn identifier]];
    return;
}
```

Finally, **numberOfRowsInTableView:** simply returns the count of the NSArray:

```
- (int)numberOfRowsInTableView:(NSTableView *)aTableView
{
    return [records count];
}
```

In each case, the NSTableView that sends the message is provided as *aTableView*. A data source object that manages several sets of data can choose the appropriate set based on which NSTableView sends the message.

Method Types

Getting values

– numberOfRowsInTableView:
– tableView:objectValueForTableColumn:row:

Setting values

– tableView:setObjectValue:forTableColumn:row:

nstance Methods

numberOfRowsInTableView:

– (int)**numberOfRowsInTableView:**(NSTableView *)*aTableView*

Returns the number of records managed for *aTableView* by the data source object. An NSTableView uses this method to determine how many rows it should create and display.

tableView:objectValueForTableColumn:row:

– (id)**tableView:**(NSTableView *)*aTableView*
objectValueForTableColumn:(NSTableColumn *)*aTableColumn*
row:(int)*rowIndex*

Returns an attribute value for the record in *aTableView* at *rowIndex*. *aTableColumn* contains the identifier for the attribute, which you get by using NSTableColumn's **identifier** method. For example, if *aTableColumn* stands for the city that an employee lives in and *rowIndex* specifies the record for an employee who lives in Portland, this method returns an object with a string value of "Portland". See the category description for an example.

tableView:setObjectValue:forTableColumn:row:

– (void)**tableView:**(NSTableView *)*aTableView*
setObjectValue:(id)*anObject*
forTableColumn:(NSTableColumn *)*aTableColumn*
row:(int)*rowIndex*

Sets an attribute value for the record in *aTableView* at *rowIndex*. *anObject* is the new value, and *aTableColumn* contains the identifier for the attribute, which you get by using NSTableColumn's **identifier** method. See the category description for an example.

NSTextAttachmentCell

Adopted By: NSTextAttachmentCell

Declared In: AppKit/NSTextAttachment.h

Protocol Description

The NSTextAttachmentCell protocol declares the interface for objects that draw text attachment icons and handle mouse events on their icons. With the exceptions of **cellBaselineOffset:**, **setAttachment:** and **attachment**, all of these methods are implemented by the NSCell class and described in that class specification.

See the NSAttributedString and NSTextView class specifications for general information on text attachments.

Method Types

Drawing

– drawWithFrame:inView:
– highlight:withFrame:inView:

Cell size and position

– cellSize
– cellBaselineOffset

Event handling

– wantsToTrackMouse
– trackMouse:inRect:ofView:untilMouseUp:

Setting the attachment

– setAttachment:
– attachment

nstance Methods

attachment

– (NSTextAttachment *)**attachment**

Returns the text attachment object that owns the receiver.

See also: – **setAttachment:**

cellBaselineOffset

– (NSPoint)**cellBaselineOffset**

Returns the position where the attachment cell's image should be drawn in text, relative to the current point established in the glyph layout. The image should be drawn so that its lower left corner lies on this point.

See also: – **icon** (NSFileWrapper)

cellSize

– (NSSize)**cellSize**

Returns the size of the attachment's icon.

See also: – **icon** (NSFileWrapper), – **fileWrapper** (NSTextAttachment)

drawWithFrame:inView:

– (void)**drawWithFrame:**(NSRect)*cellFrame* **inView:**(NSView *)*aView*

Draws the receiver's image within *cellFrame* in *aView*, which should be the focus view.

See also: – **drawWithFrame:inView:** (NSCell), – **lockFocus** (NSView)

highlight:withFrame:inView:

– (void)**highlight:**(BOOL)*flag*
withFrame:(NSRect)*cellFrame*
inView:(NSView *)*aView*

Draws the receiver's image—with highlighting if *flag* is YES—within *cellFrame* in *aView*, which should be the focus view.

See also: – **highlight:withFrame:inView:** (NSCell), – **lockFocus** (NSView)

setAttachment:

– (void)**setAttachment:**(NSTextAttachment *)*anAttachment*

Sets the text attachment object that owns the receiver to *anAttachment*, without retaining it (the text attachment, as the owner, retains the cell).

See also: – **attachment**, – **setAttachmentCell:** (NSTextAttachment)

trackMouse:inRect:ofView:untilMouseUp:

– (BOOL)**trackMouse:**(NSEvent *)*theEvent*
inRect:(NSRect)*cellFrame*
ofView:(NSView *)*aTextView*
untilMouseUp:(BOOL)*flag*

Handles a mouse-down event on the receiver's image. *theEvent* is the mouse-down event. *cellFrame* is the region of *aTextView* in which further mouse events should be tracked. *aTextView* is the view which received the event. It's assumed to be an NSTextView, and should be the focus view. If *flag* is YES, the receiver tracks the mouse until a mouse-up event occurs; if *flag* is NO, it stops tracking when a mouse-dragged event occurs outside of *cellFrame*. Returns YES if the receiver successfully finished tracking the mouse (typically through a mouse-up event), NO otherwise (such as when the mouse is dragged outside *cellFrame*).

NSTextAttachmentCell's implementation of this method calls upon *aTextView*'s delegate to handle the event. If *theEvent* is a mouse-up event for a double click, the text attachment cell sends the delegate a **textView:doubleClickedOnCell:inRect:** message and returns YES. Otherwise, depending on whether the user clicks or drags the cell, it sends the delegate a **textView:clickedOnCell:inRect:** or a **textView: draggingCell:inRect:event:** message and returns YES. NSTextAttachmentCell's implementation returns NO only if *flag* is NO and the mouse is dragged outside of *cellFrame*. The delegate methods are invoked only if the delegate responds.

See also: – **wantsToTrackMouse**, – **trackMouse:inRect:ofView:untilMouseUp:** (NSCell),
– **lockFocus** (NSView)

wantsToTrackMouse

– (BOOL)**wantsToTrackMouse**

Returns YES if the receiver will handle a mouse event occurring over its image (to support dragging, for example), NO otherwise. NSTextAttachmentCell's implementation of this method returns YES. The NSView containing the cell should invoke this method before sending a **trackMouse:inRect:ofView: untilMouseUp:** message.

For an attachment in an attributed string, if the attachment cell returns NO its attachment character should be selected rather than the cell being asked to track the mouse. this results in the attachment icon behaving as any regular glyph in text.

NSTextInput

Adopted By: NSInputManager

Declared In: AppKit/NSInputManager.h

Protocol Description

Note: *This class specification is incomplete and has not received a technical review. It is included in this release to test the linkage between the application development tools and the on-line documentation. What information it contains should be considered preliminary and subject to change.*

The methods of the NSTextInput protocol are implemented by objects—for example, responders—that handle text input. A client object such as a text view (“the client object”) that speaks this protocol must be reponsible for the following things:

1. The client object must maintain a “marked region” within which text input and, possibly, character conversion take place. The marked region may have a length of zero. The client object must maintain an “insertion point”—typically at the end of the marked region, though it may be within the region. The “selection” within the client object, if any, is entirely contained within the marked region whenever there is a marked region.2. The client object is responsible for sending messages to currentInputManager when the mouse goes down inside the marked region, or when the mouse leaves the marked region. Within the marked region, this allows the selection to be changed. Out of the marked region, it allows the region to be “abandoned”. (See below.)

3. When there is a non-zero marked region, the client object is responsible for notifying the input manager when the selection changes, or when other programmatic changes to the text affect the marked region. It can do this by sending a message to set the selection, abandon the marked region, etc.

4. When the client object relinquishes first responder, it will typically send markedTextWillBeAbandoned: to the currentInputManager. It must send markedTextWillBeAbandoned: when its insertion point (or selection moves outside the marked range. The server will typically respond by simply unmarking the region, but may remove the marked region's text entirely.

Note: If this protocol is not implemented by a client object that does have a keyDown: method, then in-line input is not possible for that client object, and will have to be handled externally.

The NSTextInput protocol is implemented by a IM to receive input from the server on behalf of the current client, and otherwise mediate between the client object and the server. It then forwards the corresponding messages to the client, or gets information from the client to pass back to the server, as appropriate.

The message passing between NSApp, IM, UIobj, and Server is all synchronous. That is, e.g., when IM sends a message to Server, any reply comes back and is relayed to UIobj before the original message returns.

A key binding manager splits the stream of keyDown: messages (intercepted by NSResopnder) into commands and text. If there is an Input Manager in the loop, it will further use any of these NSTextInput messages to control the marked region.

Method Types

Marking text

– setMarkedText:selectedRange:
– getMarkedText:selectedRange:
– hasMarkedText
– unmarkText

Other

– conversationIdentifier
– doCommandBySelector:
– insertText:

Instance Methods

conversationIdentifier

– (long)**conversationIdentifier**

Returns a number used to identify the receiver's input management session to the input server.

<<more information forthcoming>>

doCommandBySelector:

– (void)**doCommandBySelector:**(SEL)*aSelector*

Attempts to invoke *aSelector* or pass the message up the responer chain. This method is invoked by an input manager in response to an **interpretKeyEvents:** message.

<<more information forthcoming>>

See also: – **interpretKeyEvents:** (NSResponder)

getMarkedText:selectedRange:

– (void)**getMarkedText:**(out NSString **)*aString* **selectedRange:**(out NSRange *)*aRange*

Returns by reference in *aString* the receiver's marked text, if any, and in *aRange* the range of the selection within *aString* (not in terms of the receiver's entire text stream).

hasMarkedText

– (BOOL)**hasMarkedText**

Returns YES if the receiver has text that's still being interpreted by the input manager, NO if it doesn't.

<<more information forthcoming>>

insertText:

– (void)**insertText:**(NSString *)*aString*

Inserts *aString* into the receiver's text stream. This method is invoked by an input manager in response to an **interpretKeyEvents:** message.

<<more information forthcoming>>

See also: – **interpretKeyEvents:** (NSResponder)

setMarkedText:selectedRange:

– (void)**setMarkedText:**(NSString *)*aString* **selectedRange:**(NSRange)*selRange*

<<forthcoming>>

unmarkText

– (void)**unmarkText**

Removes any marking from pending input text, and accepts the text in its current state.

<<more information forthcoming>>

Defined Types

Note: This section has not been updated and has not received recent technical review. It is included in this release to test the linkage between application development tools and on-line documentation. The information in this section should be considered at best preliminary and subject to change. An updated version of this file will be included in the next release.

DPSContextRec

DECLARED IN dpsclient/dpsfriends.h

SYNOPSIS

```
typedef struct _t_DPSContextRec {
        char *priv;
        DPSSpace space;
        DPSProgramEncoding programEncoding;
        DPSNameEncoding nameEncoding;
        struct _t_DPSProcsRec const * procs;
        void (*textProc)();
        void (*errorProc)();
        DPSResults resultTable;
        unsigned int resultTableLength;
        struct _t_DPSContextRec *chainParent, *chainChild;
        DPSContextType type;
} DPSContextRec, *DPSContext;
```

DESCRIPTION The **DPSContextRec** structure represents a Display PostScript context.

DPSContextType

DECLARED IN dpsclient/dpsfriends.h

SYNOPSIS

typedef enum {
    dps_machServer,
    dps_fdServer,
    dps_stream
} **DPSContextType**;

DESCRIPTION These represent the context types supported by NeXT's version of Display PostScript, as used in the **type** field of a **DPSContextRec** structure.

DPSErrorCode

DECLARED IN dpsclient/dpsclient.h

SYNOPSIS

typedef enum _DPSErrorCode {
    dps_err_ps = DPS_ERROR_BASE,
    dps_err_nameTooLong,
    dps_err_resultTagCheck,
    dps_err_resultTypeCheck,
    dps_err_invalidContext,
    dps_err_select = DPS_NEXT_ERROR_BASE,
    dps_err_connectionClosed,
    dps_err_read,
    dps_err_write,
    dps_err_invalidFD,
    dps_err_invalidTE,
    dps_err_invalidPort,
    dps_err_outOfMemory,
    dps_err_cantConnect
} **DPSErrorCode**;

DESCRIPTION Error codes passed to a **DPSErrorProc()** function.

DPSEventFilterFunc

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS typedef int (*__DPSEventFilterFunc__)(NXEvent **ev*);

DESCRIPTION Call-back function used to filter events.

DPSFDProc

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS typedef void (*__DPSFDProc__)(int *fd*, void **userData*);

DESCRIPTION Call-back function used when a file descriptor is registered through **DPSAddFD()**.

DPSNumberFormat

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS

```
typedef enum _DPSNumberFormat {
#ifdef __BIG_ENDIAN__
        dps_float = 48,
        dps_long = 0,
        dps_short = 32
#else
        dps_float = 48+128,
        dps_long = 0+128,
        dps_short = 32+128
} DPSNumberFormat;
```

DESCRIPTION These constants are used by the **DPSDoUserPath()** function to describe the type of numbers that are being passed.

DPSPingProc

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS

typedef void (*__DPSPingProc__)
(DPSContext *ctxt*,
void **userData*);

DESCRIPTION Call-back function used by **DPSAsynchronousWaitContext()**.

DPSPortProc

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS

typedef void (*__DPSPortProc__)
(msg_header_t **msg*,
void **userData*);

DESCRIPTION Call-back function used when a port is registered through **DPSAddPort()**.

DPSTimedEntry

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS

typedef struct __DPSTimedEntry ***DPSTimedEntry**;

DESCRIPTION The return type for **DPSAddTimedEntry()**.

DPSTimedEntryProc

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS

typedef void (*__DPSTimedEntryProc__)
(DPSTimedEntry *timedEntry*,
double *now*,
void **userData*);

DESCRIPTION Call-back function used when a timed entry is registered through **DPSAddTimedEntry()**.

DPSUserPathAction

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS

typedef enum _DPSUserPathAction {
dps_uappend,
dps_ufill,
dps_ueofill,
dps_ustroke,
dps_ustrokepath,
dps_inufill,
dps_inueofill,
dps_inustroke,
dps_def,
dps_put
} **DPSUserPathAction**;

DESCRIPTION These constants are convenient representations of some of the PostScript operator indices, suitable for enrollment in the action array passed to **DPSDoUserPath()**.

DPSUserPathOp

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS

```
typedef enum _DPSUserPathOp {
        dps_setbbox,
        dps_moveto,
        dps_rmoveto,
        dps_lineto,
        dps_rlineto,
        dps_curveto,
        dps_rcurveto,
        dps_arc,
        dps_arcn,
        dps_arct,
        dps_closepath,
        dps_ucache
} DPSUserPathOp;
```

DESCRIPTION These constants represent the PostScript operators that can be passed in **DPSDoUserPath()**'s operator array.

NXCoord

DECLARED IN dpsclient/event.h

SYNOPSIS typedef float **NXCoord**

DESCRIPTION Used to represent a single coordinate in a Cartesian coordinate system.

NXEvent

DECLARED IN dpsclient/event.h

SYNOPSIS

```
typedef struct _NXEvent {
        int type;
        NXPoint location;
        long time;
        int flags;
        unsigned int window;
        NXEventData data;
        DPSContext ctxt;
} NXEvent, *NXEventPtr;
```

DESCRIPTION Represents a single event; this structure is also known as the *event record*. The fields are:

type	The type of event (see “Event Types,” below)
location	The event’s location in the base coordinate system of its window
time	The time of the event (in hardware-dependent units) since system startup
flags	Mouse-button and modifier-key flags (see “Event Flags,” below)
window	The window number of the window associated with the event
data	Additional type-specific data (see “NXEventData,” below)
ctxt	The PostScript context of the event

NXEventData

DECLARED IN dpsclient/event.h

SYNOPSIS

```
typedef union {
    struct {
        short eventNum;
        int click;
        unsigned char pressure;
    } mouse;
    struct {
        short repeat;
        unsigned short charSet;
        unsigned short charCode;
        unsigned short keyCode;
        short keyData;
    } key;
    struct {
        short eventNum;
        int trackingNum;
        int userData;
    } tracking;
    struct {
        short subtype;
        union {
            float F[2];
            long L[2];
            short S[4];
            char C[8];
        } misc;
    } compound;
} NXEventData;
```

DESCRIPTION This structure supplies type-specific information for an event. It's a union of four structures, where the type of the event determines which structure is pertinent:

- **mouse** is used for mouse events.
- **key** is used for keyboard events.
- **tracking** is for tracking-rectangle events.
- **compound** is for system-, kit-, and application-defined events.

NXPoint

DECLARED IN dpsclient/event.h

SYNOPSIS

```
typedef struct _NXPoint {
        NXCoord x;
        NXCoord y;
} NXPoint;
```

DESCRIPTION Represents a point in a Cartesian coordinate system.

NXSize

DECLARED IN dpsclient/event.h

SYNOPSIS

```
typedef struct _NXSize {
        NXCoord width;
        NXCoord height;
} NXSize;
```

DESCRIPTION Represents a two-dimensional size.

Symbolic Constants

All Contexts

DECLARED IN dpsclient/NSDPSContext.h

SYNOPSIS DPS_ALLCONTEXTS

DESCRIPTION This constant represents all extant contexts.

Alpha Constants

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS
NX_DATA
NX_ONES

DESCRIPTION These constants represent alpha values.

Character Set Values

DECLARED IN dpsclient/event.h

SYNOPSIS
NX_ASCIISET
NX_SYMBOLSET
NX_DINGBATSSET

DESCRIPTION These constants represent the values that may occur in the **data.key.charSet** field of an NXEvent structure.

Compositing Operations

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS

NX_CLEAR
NX_COPY
NX_SOVER
NX_SIN
NX_SOUT
NX_SATOP
NX_DOVER
NX_DIN
NX_DOUT
NX_DATOP
NX_XOR
NX_PLUSD
NX_HIGHLIGHT
NX_PLUSL

DESCRIPTION These represent the compositing operations used by **PScomposite()** and the NXImage class.

Error Code Bases

DECLARED IN dpsclient/dpsclient.h

SYNOPSIS

DPS_ERROR_BASE
DPS_NEXT_ERROR_BASE

DESCRIPTION These constants represent the lowest values for Display PostScript error codes.

Event Types

DECLARED IN dpsclient/event.h

Type	Meaning
NX_NULLEVENT	A non-event
NX_LMOUSEDOWN	Left mouse-down
NX_LMOUSEUP	Left mouse-up
NX_LMOUSEDRAGGED	left mouse-dragged
NX_MOUSEDOWN	Same as NX_LMOUSEDOWN
NX_MOUSEUP	Same as NX_LMOUSEUP
NX_MOUSEDRAGGED	Same as NX_LMOUSEDRAGGED
NX_RMOUSEDOWN	Right mouse-down
NX_RMOUSEUP	Right mouse-up
NX_RMOUSEDRAGGED	Right mouse-dragged
NX_MOUSEMOVED	Mouse-moved
NX_MOUSEENTERED	Mouse-entered
NX_MOUSEEXITED	Mouse-exited
NX_KEYDOWN	Key-down
NX_KEYUP	Key-up event
NX_FLAGSCHANGED	Flags-changed
NX_KITDEFINED	Application Kit-defined
NX_SYSDEFINED	System-defined
NX_APPDEFINED	Application-defined
NX_TIMER	Timer used for tracking
NX_CURSORUPDATE	Cursor tracking
NX_JOURNALEVENT	Event used by journaling
NX_FIRSTEVENT	The smallest-valued event constant
NX_LASTEVENT	The greatest-valued event constant
NX_ALLEVENTS	A value that includes all event types

DESCRIPTION These constants represent event types. They're passed as the **type** field of the NXEvent structure that's created when an event occurs.

Event Type Masks

DECLARED IN dpsclient/event.h

SYNOPSIS

```
NX_NULLEVENTMASK
NX_LMOUSEDOWNMASK
NX_LMOUSEUPMASK
NX_RMOUSEDOWNMASK
NX_RMOUSEUPMASK
NX_MOUSEMOVEDMASK
NX_LMOUSEDRAGGEDMASK
NX_RMOUSEDRAGGEDMASK
NX_MOUSEENTEREDMASK
NX_MOUSEEXITEDMASK
NX_KEYDOWNMASK
NX_KEYUPMASK
NX_FLAGSCHANGEDMASK
NX_KITDEFINEDMASK
NX_APPDEFINEDMASK
NX_SYSDEFINEDMASK
NX_TIMERMASK
NX_CURSORUPDATEMASK
NX_MOUSEDOWNMASK
NX_MOUSEUPMASK
NX_MOUSEDRAGGEDMASK
NX_JOURNALEVENTMASK
```

DESCRIPTION These masks correspond to the event types defined immediately above. They let you query the **type** field of an NXEvent structure for the existence of a particular event type.

Forever

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS NX_FOREVER

DESCRIPTION A long, long time. Typically used as the timeout argument to **DPSGetEvent()**.

Keyboard State Flags Masks

DECLARED IN dpsclient/event.h

Type	Meaning
NX_ALPHASHIFTMASK	Shift lock
NX_SHIFTMASK	Shift key
NX_CONTROLMASK	Control key
NX_ALTERNATEMASK	Alt key
NX_COMMANDMASK	Command key
NX_NUMERICPADMASK	Number pad key
NX_HELPMASK	Help key
NX_NEXTCTRLKEYMASK	Control key
NX_NEXTLSHIFTKEYMASK	Left shift key
NX_NEXTRSHIFTKEYMASK	Right shift key
NX_NEXTLCMDKEYMASK	Left command key
NX_NEXTRCMDKEYMASK	Right command key
NX_NEXTLALTKEYMASK	Left alt key
NX_NEXTRALTKEYMASK	Right alt key

DESCRIPTION These masks correspond to keyboard states that might be included in an NXEvent structure's **flags** mask. The masks are grouped as device-independent (NX_ALPHASHIFTMASK through NX_HELPMASK) and device-dependent (all others).

Miscellaneous Event Flags Masks

DECLARED IN dpsclient/event.h

Type	Meaning
NX_STYLUSPROXIMITYMASK	Stylus is in proximity (for tablets)
NX_NONCOALSESCEDMASK	Event coalescing disabled

DESCRIPTION These masks correspond to miscellaneous states that might be included in an NXEvent structure's **flags** mask.

Window Backing Types

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS

NX_RETAINED
NX_NONRETAINED
NX_BUFFERED

DESCRIPTION These represent the three backing types provided by window devices (and used by the Application Kit's Window objects).

Window Screen List Placement

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS

NX_ABOVE
NX_BELOW
NX_OUT

DESCRIPTION These represent the placement of a window device in the screen list.