NeXTstep Reference Volume 2

NeXT Developer's Library

NeXTstep

Draw upon the library of software contained in NeXTstep to develop your applications. Integral to this development environment are the Application Kit and Display PostScript.

Concepts

A presentation of the principles that define NeXTstep, including user interface design, object-oriented programming, event handling, and other fundamentals.

Reference, Volumes 1 and 2

Detailed, comprehensive descriptions of the NeXTstep Application Kit software.

Sound, Music, and Signal Processing

Let your application listen, talk, and sing by using the Sound Kit and the Music Kit. Behind these capabilities is the DSP56001 digital signal processor. Independent of sound and music, scientific applications can take advantage of the speed of the DSP.

Concepts

An examination of the design of the sound and music software, including chapters on the use of the DSP for other, nonaudio uses.

Reference

Detailed, comprehensive descriptions of each piece of the sound, music, and DSP software.

NeXT Development Tools

A description of the tools used in developing a NeXT application, including the Edit application, the compiler and debugger, and some performance tools.

NeXT Operating System Software

A description of NeXT's operating system, Mach. In addition, other low-level software is discussed.

Writing Loadable Kernel Servers

How to write loadable kernel servers, such as device drivers and network protocols.

NeXT Technical Summaries

Brief summaries of reference information related to NeXTstep, sound, music, and Mach, plus a glossary and indexes.

Supplemental Documentation

Information about PostScript, RTF, and other file formats useful to application developers.

NeXTstep Reference
Volume 2

Manual written by Don Larkin, Matt Morse, Jim Inscore, Sam Streeper, and Jackie Neider
Edited by Caroline Rose, Kathy Walrath, Roy West, Helen Casabona, Adrienne Wong, and Jeremy Brest
Book design by Eddie Lee
Illustrations by Jeff Yaksick and Don Donoughe
Production by Adrienne Wong, Jennifer Yu, and Katherine Arthurs
Publications management by Cathy Novak

Reorder Product #N6007B

Contents

Chapter 2
Class Specifications

PageLayout

INHERITS FROM	Panel : Window : Responder : Object
DECLARED IN	appkit/PageLayout.h

CLASS DESCRIPTION

PageLayout is a type of Panel that queries the user for information such as paper type and orientation. This information is passed to the Application object's PrintInfo object, and is later used when printing. You can invoke the **setAccessoryView:** method to add your own View to the PageLayout panel to extend its functionality. An application can have only one PageLayout object; the **new** method returns the previous instance of the PageLayout object if one already exists. Most applications will bring up this panel by invoking the Application method **runPageLayout:** (this method is sent up the responder chain when you click the Page Layout menu item), but you can also run the panel with the PageLayout method **runModal**.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Responder	id	nextResponder;
Inherited from Window	NXRect	frame;
	id	contentView;
	id	delegate;
	id	firstResponder;
	id	lastLeftHit;
	id	lastRightHit;
	id	counterpart;
	id	fieldEditor;
	int	winEventMask;
	int	windowNum;
	float	backgroundGray;
	struct _wFlags	wFlags;
	struct _wFlags2	wFlags2;
Inherited from Panel	(none)	

Declared in PageLayout	id	appIcon;
	id	height;
	id	width;
	id	ok;
	id	cancel;
	id	orientation;
	id	scale;
	id	paperSizeList;
	id	layoutList;
	id	unitsList;
	int	exitTag;
	id	paperView;
	id	accessoryView;

appIcon	The Button object with the Application's icon.
height	The Form object for paper height.
width	The Form object for paper width.
ok	The OK Button object.
cancel	The Cancel Button object.
orientation	The Matrix object for choosing between portrait and landscape orientation.
scale	The TextField for the scaling factor.
paperSizeList	The Button object for the PopUpList of paper choices.
layoutList	The Button object for the PopUpList of layout choices.
unitsList	The Button object for the PopUpList of unit choices.
exitTag	The tag of the Button object the user clicked to exit the Panel.
paperView	The View used to display the size and orientation of the selected paper type. A subclass could set this instance to its own View to display a customized paper representation.
accessoryView	The optional View added by the application.

METHOD TYPES

Creating and freeing an instance	+ new + newContent:style:backing:buttonMask:defer: – free
Running the PageLayout panel	– runModal
Customizing the PageLayout panel	– setAccessoryView: – accessoryView
Updating the panel's display	– pickedLayout: – pickedOrientation: – pickedPaperSize: – pickedUnits: – textDidEnd:endChar: – textWillChange: – convertOldFactor:newFactor: – pickedButton:
Communicating with the PrintInfo object	– readPrintInfo – writePrintInfo

CLASS METHODS

alloc

Generates an error message. This method cannot be used to create PageLayout instances. Use **new** instead.

See also: + **new**

allocFromZone:

Generates an error message. This method cannot be used to create PageLayout instances. Use **new** instead.

See also: + **new**

new

+ new

Creates and returns the Page Layout panel. This will return the existing instance of the Page Layout panel if one has already been created.

newContent:style:backing:buttonMask:defer:

+ **newContent:**(const NXRect *)*contentRect*
  style:(int)*aStyle*
  backing:(int)*bufferingType*
  buttonMask:(int)*mask*
  defer:(BOOL)*flag*

Used in the instantiation of the Page Layout panel. You shouldn't use this method to create the panel; use **new** instead.

See also: + **new**

INSTANCE METHODS

accessoryView

– **accessoryView**

Returns the custom accessory View set by **setAccessoryView:**.

See also: – **setAccessoryView:**

convertOldFactor:newFactor:

– **convertOldFactor:**(float *)*old* **newFactor:**(float *)*new*

Returns conversion factors for values displayed in the panel. If this method is invoked from within an override of the **pickedUnits:** method, it will set **old** to the conversion factor between the unit of points and the previous units selected; **new** will be set to the conversion factor between points and the new units just selected. If this method is invoked at other times, such as when the page layout information is being loaded with the **readPrintInfo** method, both **old** and **new** will be set to the conversion factor between points and the currently selected units. See **pickedUnits:** for an example. Returns **self**.

See also: – **pickedUnits:**

free

– **free**

Frees all the memory used by the Page Layout panel.

See also: + **new**

pickedButton:

– **pickedButton:***sender*

Ends the current run of the Page Layout panel if all the entries in the panel are valid. If the entries are not valid, this method does nothing. This method is the target of the OK and Cancel buttons. If all the panel entries are valid, this method sets the **exitTag** instance variable to the tag of the button that the user clicked to dismiss the panel, and sends a **stopModal** message to the Application object. Returns **self**.

See also: – **runModal**, – **stopModal** (Application)

pickedLayout:

– **pickedLayout:***sender*

Performed when the user selects an item from the layout list. You might override this method to update other controls you add to the panel. You can get the new layout with the message [[sender selectedCell] title]. Returns **self**.

See also: – **setAccessoryView:**, – **selectedItem** (PopUpList), – **selectedCell** (Matrix)

pickedOrientation:

– **pickedOrientation:***sender*

Performed when the user selects a page orientation. This method updates the paper width and height forms. You can override it to update other controls you add to the panel. You can get the new orientation with the message [sender selectedCol], where a return value of 0 means portrait, and a value of 1 means landscape. Returns **self**.

See also: – **setAccessoryView:**, – **selectedCol** (Matrix)

pickedPaperSize:

– **pickedPaperSize:***sender*

Performed when the user selects a paper size. This method updates the paper width and height forms, and may switch the page orientation. You can override this method to update other controls you add to the panel. You can get the new name of the new paper size with the message [[sender selectedCell] title]. Returns **self**.

See also: – **setAccessoryView:**, – **selectedItem** (PopUpList), – **selectedCell** (Matrix)

pickedUnits:

– pickedUnits:*sender*

Performed when the user selects a new unit of measurement. You can override this method to update other controls you add to the panel. You should do this for any fields you add that express dimensions on the page. To determine how to update your field, call the PageLayout method **convertOldFactor:newFactor:**. The first value will convert from the unit of points to the previous unit of measurement. The second will convert from points to the new unit of measurement. The following example supposes that a subclass of PageLayout adds a TextField stored in the instance variable myField:

```
- pickedUnits:sender
{
    float old, new;

    /*  At this point, the units have been selected, */
    /*  but not set. Get the conversion factors: */

    [self convertOldFactor:&old newFactor:&new];
    /*  Set my field based on the conversion factors  */
    [myField setFloatValue:([myField floatValue] * new / old)];

    /*  Now let the method set the selected units  */
    return [super pickedUnits:sender];
}
```

See also: **– convertOldFactor:newFactor:**, **– setAccessoryView:**

readPrintInfo

– readPrintInfo

Reads the Application's global PrintInfo object, and sets the values of the Page Layout panel to those in the PrintInfo. This method is invoked from the **runModal** method; you should not need to invoke it yourself. Returns **self**.

See also: **– writePrintInfo**, **– runModal**

runModal

– (int)**runModal**

Runs the Page Layout panel. For most applications, this is the only method needed to use this object. It loads the current printing information into the panel from the Application's global PrintInfo object. It then runs the panel using Application's **runModalFor:** method. When the user finishes with the panel, it is hidden. If the user exited the panel via the OK button, the information that he filled in is written back to the global PrintInfo object. The method returns the tag of the button that the user chose to dismiss the panel (either NX_OKTAG or NX_CANCELTAG). Note that since **runModalFor:** is used to run the Page Layout panel, the **pickedButton:** method must terminate the modal run by invoking Application's **stopModal** method.

See also: – **runPageLayout** (Application), – **pickedButton:**,
– **stopModal** (Application), – **runModalFor:** (Application)

setAccessoryView:

– **setAccessoryView:***aView*

Adds *aView* to the contents of the Page Layout panel. Applications can invoke this method to add controls to extend the functionality of the panel. *aView* should be the top View in a View hierarchy. The Page Layout panel 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 any accessory View that's in the panel is removed. Returns the old accessory View.

See also: – **accessoryView**

textDidEnd:endChar:

– **textDidEnd:***textObject* **endChar:**(unsigned short)*theChar*

Performed when user finishes typing a page size. Selects the correct orientation to match the new paper size. You can override this method to update other controls you add to the panel. The width and height fields are Form objects, so you can use the Form method **floatValueAt:**0 to get the values of these fields. Returns **self**.

See also: – **setAccessoryView:**, – **floatValueAt:** (Form)

textWillChange:

– (BOOL)**textWillChange:***textObject*

Performed when the user types in a page size. This method highlights the "Other" choice in the list of paper types. You can override this method to update other controls you add to the panel. This message is sent to the PageLayout object because it is the Text object's delegate; it returns 0 to indicate that the text field can be changed.

See also: – **setAccessoryView:**, – **textWillChange:** (Text delegate)

writePrintInfo

– **writePrintInfo**

Writes the settings of the Page Layout panel to the Application object's global PrintInfo object. This method is invoked when the user quits the Page Layout panel by clicking the OK button. Returns **self**.

See also: – **readPrintInfo**, – **runModal**

CONSTANTS AND DEFINED TYPES

```
/* Tags of Controls in the Page Layout panel */
#define NX_PLICONBUTTON       50
#define NX_PLTITLEFIELD       51
#define NX_PLPAPERSIZEBUTTON  52
#define NX_PLLAYOUTBUTTON     53
#define NX_PLUNITSBUTTON      54
#define NX_PLWIDTHFORM        55
#define NX_PLHEIGHTFORM       56
#define NX_PLPORTLANDMATRIX   57
#define NX_PLSCALEFIELD       58
#define NX_PLCANCELBUTTON     NX_CANCELTAG
#define NX_PLOKBUTTON         NX_OKTAG
```

Panel

INHERITS FROM	Window : Responder : Object
DECLARED IN	appkit/Panel.h

CLASS DESCRIPTION

A Panel is a Window that serves an auxiliary function within an application; it contains Views that give information to users and let users give instructions to the application. Usually, the Views are Control objects of some sort—Buttons, Forms, NXBrowsers, TextViewers, Sliders, and so on. Menu is a Panel subclass.

Panels behave differently than other Windows in only a small number of ways, but the ways are important to the user interface:

- Panels pass Command key-down events to the objects in their view hierarchies. This permits them to have keyboard alternatives.
- Panels aren't destroyed when closed; they're simply moved off-screen (taken out of the screen list).
- On-screen Panels are removed from the screen list when the user begins to work in another application, and are restored to the screen when the user returns to the Panel's application.
- Panels have a light gray, rather than white, background in their content area.

To facilitate their intended roles in the user interface, some panels can be assigned special behaviors:

- A panel can be precluded from becoming the key window until the user makes a selection (makes a View the first responder) indicating an intention to begin typing. This prevents key window status from shifting to the Panel unnecessarily.
- 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.
- A Panel can be made to work—to receive mouse and keyboard events—even when there's an attention panel on-screen. This permits actions within the Panel to affect the attention panel.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Responder	id	nextResponder;
Inherited from Window	NXRect	frame;
	id	contentView;
	id	delegate;
	id	firstResponder;
	id	lastLeftHit;
	id	lastRightHit;
	id	counterpart;
	id	fieldEditor;
	int	winEventMask;
	int	windowNum;
	float	backgroundGray;
	struct _wFlags	wFlags;
	struct _wFlags2	wFlags2;
Declared in Panel	(none)	

METHOD TYPES

Initializing a new Panel	– init
	– initContent:style:backing:buttonMask:defer:
Handling events	– commandKey:
	– keyDown:
Determining the Panel interface	– setBecomeKeyOnlyIfNeeded:
	– doesBecomeKeyOnlyIfNeeded
	– setFloatingPanel:
	– isFloatingPanel
	– setWorksWhenModal:
	– worksWhenModal

INSTANCE METHODS

commandKey:

– (BOOL)**commandKey:**(NXEvent *)*theEvent*

Intercepts **commandKey:** messages being passed from Window to Window, and translates them to **performKeyEquivalent:** messages for the Views within the Panel. This method returns YES if any of the Views can handle the event as its keyboard alternative, and NO if none of them can. A NO return continues the **commandKey:** message down the Application object's list of windows; a YES return terminates it.

The Application object initiates **commandKey:** messages when it gets key-down events with the Command key pressed. The Panel also initiates them, but just to itself, when it gets a **keyDown:** event message. The argument, *theEvent*, is a pointer to the key-down event.

Before any **performKeyEquivalent:** messages are sent, a Panel that's not on-screen receives an **update** message. This gives it a chance to make sure that its Views are properly enabled or disabled to reflect the current state of the application.

See also: – **keyDown:**, – **performKeyEquivalent:** (View)

doesBecomeKeyOnlyIfNeeded

– (BOOL)**doesBecomeKeyOnlyIfNeeded**

Returns whether the Panel refrains from becoming the key window until the user clicks within (sends a mouse-down event to) a View that can become the first responder. The default is NO.

See also: – **setBecomeKeyOnlyIfNeeded:**

init

– **init**

Initializes the receiver, a newly allocated Panel object, by sending it an **initContent:style:backing:buttonMask:defer:** message with default parameters, and returns **self**.

The Panel will have a content rectangle of minimal size. The Window Server won't create a window for the Panel until the Panel is ready to be displayed on-screen; the window will be a buffered window. The Panel will have a title bar and close button, but no resize bar. Like all Windows, it's initially placed out of the screen list. Its title is not set.

See also: – **initContent:style:backing:buttonMask:defer:**

initContent:style:backing:buttonMask:defer:

– **initContent:**(const NXRect *)*contentRect*
  style:(int)*aStyle*
  backing:(int)*bufferingType*
  buttonMask:(int)*mask*
  defer:(BOOL)*flag*

Initializes the receiver, a newly allocated Panel instance, and returns **self**.

This method is the designated initializer for this class. It's identical to the Window method of the same name, except that it additionally initializes the receiver so that it will behave like a panel in the user interface:

- The Panel's background color is set to be light gray.
- The Panel will hide when the application it belongs to is deactivated.
- The Panel won't be freed when the user closes it.

The new Panel is initially out of the Window Server's screen list. To make it visible, you must **display** it (into the buffer) and then move it on-screen.

See also: – **initContent:style:backing:buttonMask:defer:** (Window)

isFloatingPanel

– (BOOL)**isFloatingPanel**

Returns whether the Panel floats above standard windows and other panels. The default is NO.

See also: – **setFloatingPanel:**

keyDown:

– **keyDown:**(NXEvent *)*theEvent*

Translates the key-down event into a **commandKey:** message for the Panel, thus interpreting the event as a potential keyboard alternative. If the Panel has a button that displays the Return symbol and the key-down event is for the Return key, it will operate the button.

A Panel can receive **keyDown:** event messages only when it's the key window and none of its Views is the first responder.

See also: – **commandKey:**

setBecomeKeyOnlyIfNeeded:

– **setBecomeKeyOnlyIfNeeded:**(BOOL)*flag*

Sets whether the Panel becomes the key window only when the user makes a selection (causing one of its Views to become the first responder). Since this requires the user to perform an extra action (clicking in the View) before being able to type within the window, it's appropriate only for Panels that don't normally require text entry. You should consider setting this attribute only if (1) most of the controls within the Panel are not text fields, and (2) the choices that can be made by entering text can also be made in another way (or are only incidental to the way the panel is normally used). The default *flag* is NO. Returns **self**.

See also: – **doesBecomeKeyOnlyIfNeeded**, – **keyDown:**

setFloatingPanel:

– **setFloatingPanel:**(BOOL)*flag*

Sets whether the Panel should be assigned to a window tier above standard windows. The default is NO. It's appropriate for a Panel to float above other windows only if:

- It's oriented to the mouse rather than the keyboard—that is, it doesn't become the key window (or becomes the key window only if needed),
- It needs to remain visible while the user works in the application's standard windows—for example, if the user must frequently move the cursor back and forth between a standard window and the panel (such as a tool palette) or the panel gives information relevant to the user's actions within a standard window,
- It's small enough not to obscure much of what's behind it, and
- It doesn't remain on-screen when the application is deactivated.

All four of these test must be met for *flag* to be set to YES. Returns **self**.

See also: – **isFloatingPanel**

setWorksWhenModal:

– **setWorksWhenModal:**(BOOL)*flag*

Sets whether the Panel remains enabled to receive events and possibly become the key window even when a modal panel (attention panel) is on-screen. This is appropriate only for a Panel that needs to operate on attention panels. The default is NO. Returns **self**.

See also: – **worksWhenModal**

worksWhenModal

– (BOOL)**setWorksWhenModal**

Returns whether the Panel can receive keyboard and mouse events and possibly become the key window, even when a modal panel (attention panel) is on-screen. The default is NO.

See also: – **setWorksWhenModal:**

CONSTANTS AND DEFINED TYPES

```
/*
 * Values returned by NXRunAlertPanel() (also returned by
 * runModalSession: when the modal session is run with a Panel
 * provided by NXGetAlertPanel())
 */
#define NX_ALERTDEFAULT      1
#define NX_ALERTALTERNATE    0
#define NX_ALERTOTHER       -1
#define NX_ALERTERROR       -2

/*
 * Tags for common buttons in panels
 */
#define NX_OKTAG             1
#define NX_CANCELTAG         0
```

Pasteboard

INHERITS FROM	Object
DECLARED IN	appkit/Pasteboard.h

CLASS DESCRIPTION

Pasteboard objects transfer data to and from the pasteboard server, **pbs**. The server is shared by all running applications. It contains data that the user has cut or copied and may paste, as well as other data that one application wants to transfer to another. Pasteboard 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 named data is, in effect, a separate pasteboard, distinct from the others. An application keeps a separate Pasteboard object for each named pasteboard that it uses. There are four standard pasteboards in common use:

Font pasteboard	The pasteboard that holds font and character information and supports the Copy Font and Paste Font commands.
Ruler pasteboard	The pasteboard that holds information about paragraph formats in support of the Copy Ruler and Paste Ruler commands.
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 the search.
Selection 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.

Each standard pasteboard is identified by a unique name designated by a global variable of type NXAtom:

NXFontPboard
NXRulerPboard
NXFindPboard
NXSelectionPboard

You can also create private pasteboards by asking for a Pasteboard object with any other name. The name of a private pasteboard can be passed to other applications to allow them to share the data it holds.

The Pasteboard class makes sure there's just one object for each named pasteboard. If you ask for a new object when one has already been created for the pasteboard, the existing one will be returned to you. For this reason, only the **new** and **newName:** methods defined in this class should be used to create Pasteboard objects; Object's **alloc** and **allocFromZone:** methods can't be used.

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 applications as possible can make use of it.

Data types are identified by character strings containing a full type name. The following global variables are string pointers for the standard NeXT pasteboard types. They're of type NXAtom.

Type	**Description**
NXAsciiPboardType	Plain ASCII text
NXPostScriptPboardType	Encapsulated PostScript code (EPS)
NXTIFFPboardType	Tag Image File Format (TIFF)
NXRTFPboardType	Rich Text Format (RTF)
NXSoundPboardType	The Sound object's pasteboard type
NXFilenamePboardType	ASCII text designating a file name
NXTabularTextPboardType	Tab-separated fields of ASCII text
NXFontPboardType	Font and character information
NXRulerPboardType	Paragraph formatting information

Other data types can also be used. For example, your application may keep data in a private format that's richer than any of types listed above. That format can also be used as a pasteboard type.

Reading and Writing Data

The pasteboard server supports a simple interface to reading and writing data, using a pointer to the data and the length of the data in bytes. Data is written to the pasteboard using **writeType:data:length:** and read using **readType:data:length:**. In each case only a pointer to the data is passed. The pointer and a single copy of the data can be shared among many applications.

It's often convenient to prepare data for the pasteboard by opening a memory stream and writing the data to it using functions like **NXWrite()**, **NXPrintf()**, and **NXPutc()**. After the data has been written, a pointer to the data and the number of bytes can be extracted from the stream and sent to the pasteboard server. Using a stream means that the data will be page-aligned, so it will occupy the fewest number of pages possible.

Similarly, you can create a memory stream for the data received from the pasteboard server and use functions like **NXGetc()**, **NXRead()**, and **NXScanf()** to parse it. Objects can be archived to and from the pasteboard server using typed streams.

Errors

Except where errors are specifically mentioned in the method descriptions, any communications error with the pasteboard server causes an NX_pasteboardComm exception to be raised.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Declared in Pasteboard	id	owner;
owner	The object responsible for putting data in the pasteboard.	

METHOD TYPES

Creating and freeing a Pasteboard object	+ new + newName: – free – freeGlobally
Referring to a Pasteboard by name	+ newName: – name
Writing data	– declareTypes:num:owner: – writeType:data:length:
Reading data	– changeCount – types – readType:data:length:

CLASS METHODS

alloc

Generates an error message. This method cannot be used to create Pasteboard instances. Use **new** or **newName:** instead.

See also: + **new**, + **newName:**

allocFromZone:

Generates an error message. This method cannot be used to create Pasteboard instances. Use **new** or **newName:** instead.

See also: + **new**, + **newName:**

new

+ **new**

Returns the Pasteboard object for the selection pasteboard, by passing **NXSelectionPboard** to the **newName:** method.

newName:

+ **newName:**(const char *)*name*

Returns the Pasteboard object for the *name* pasteboard. A new object is created only if the application doesn't yet have a Pasteboard object for the specified name; otherwise, the existing one is returned. To get a standard pasteboard, *name* should be one of the following variables:

NXFontPboard
NXRulerPboard
NXFindPboard
NXSelectionPboard

Other names can be assigned to create private pasteboards for other purposes.

INSTANCE METHODS

changeCount

– (int)**changeCount**

Returns the current change count of the pasteboard. The change count is a system-wide global that increments every time the contents of the pasteboard changes (a new owner is declared). It allows applications the optimization of knowing whether the current data in the pasteboard is the same as the data they last received.

An independent change count is maintained for each named pasteboard.

See also: – **declareTypes:num:owner:**

declareTypes:num:owner:

– **declareTypes:**(const char * const *)*newTypes*
    num:(int)*numTypes*
    owner:*newOwner*

Prepares the pasteboard 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:num:owner:** message is tantamount to changing the contents of the pasteboard. It invalidates the current contents of the pasteboard and increments its change count.

numTypes is the number of types the new contents of the pasteboard may assume, and *newTypes* is an array of null-terminated strings that name those types. The types should be ordered according to the preference of the source application, with the most preferred type coming first. Usually, the richest representation is the one most preferred.

The *newOwner* is the object responsible for writing data to the pasteboard in all the types listed in *newTypes*. Data is written using the **writeType:data:length:** method. 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:provideData:** 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 freed. It should not, for example, be the View that displays the data if that View is in a window that might be closed.

Returns **self**.

See also: – **writeType:data:length:**, – **pasteboard:provideData:**

free

– **free**

Frees the Pasteboard object. A Pasteboard object should not be freed if there's a chance that the application might want to use the named pasteboard again; standard pasteboards generally should not be freed at all.

freeGlobally

– **freeGlobally**

Frees the Pasteboard object and the domain for its name within the pasteboard server. This means that no other application will be able to use the named pasteboard. A temporary, privately named pasteboard can be freed when it's no longer needed, but a standard pasteboard should never be freed globally.

name

– (const char *)**name**

Returns the name of the Pasteboard object.

See also: + **newName:**

readType:data:length:

– **readType:**(const char *)*dataType*
  data:(char **)*theData*
  length:(int *)*numBytes*

Reads the *dataType* representation of the current contents of the pasteboard. *dataType* should be one of the types returned by the **types** method. The data is read by setting the pointer referred to by *theData* to the address of the data, and setting the integer referred to by *numBytes* to the length of the data in bytes.

If the data is successfully read, this method returns **self**. It returns **nil** if the contents of the pasteboard have changed (if the change count has been incremented by a **declareTypes:num: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 pasteboard's owner is slow in responding to a **pasteboard:provideData:** message and the interprocess communication times out. All other errors raise an NX_pasteboardComm 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. It should not attempt to use the pointer referred to by *theData*, as it won't be valid.

The memory for the data that this method provides is allocated directly from the Mach virtual memory manager, not through **malloc**(); it therefore should be freed only by **vm_deallocate**(), not **free**(). For example:

```
char *data;
int   length;

if ([myPasteboard readType:NXAsciiPboardType
                       data:&data length:&length])
{
    /* Use the data here, keeping it for as long as necessary */
    vm_deallocate(task_self(), data, length);
}
```

See also: – **types**

types

– (const NXAtom *)**types**

Returns the list of the types that were declared for the current contents of the pasteboard. The list is an array of character pointers holding the type names, with the last pointer being NULL. Each of the pointers is of type NXAtom, meaning that the type name is a unique string.

Types are listed in the same order that they were declared. A **types** message should be sent before reading any data from the pasteboard.

See also: – **declareTypes:num:owner:**, – **readType:data:length:**, **NXUniqueString**()

writeType:data:length:

– **writeType:**(const char *)*dataType*
 data:(const char *)*theData*
 length:(int)*numBytes*

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:num:owner:** message. *theData* points to the data to be sent to the pasteboard server, and *numBytes* is the length of the data in bytes.

A separate **writeType:data:length:** message is required for each data representation that's written to the server.

This method returns **self** if the data is successfully written. It returns **nil** if an object in another application has become the owner of the pasteboard. Any other error raises an NX_pasteboardComm exception.

See also: – **declareTypes:num:owner:**

METHODS IMPLEMENTED BY THE OWNER

pasteboard:provideData:

– **pasteboard:***sender* **provideData:**(NXAtom)*type*

Implemented by the owner (previously declared in a **declareTypes:num:owner:** message) to provide promised data. The owner receives a **pasteboard:provideData:** 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 **writeType:data:length:** method.

pasteboard:provideData: 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:provideData:** 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 pasteboard in the richest, and therefore most preferred, type at the time of a cut or copy operation, its **pasteboard:provideData:** 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.

See also: – **declareTypes:num:owner:**, – **writeType:data:length:**

CONSTANTS AND DEFINED TYPES

```
/*
 * standard Pasteboard types
 */
extern NXAtom NXAsciiPboardType;
extern NXAtom NXPostScriptPboardType;
extern NXAtom NXTIFFPboardType;
extern NXAtom NXRTFPboardType;
extern NXAtom NXFilenamePboardType;
extern NXAtom NXTabularTextPboardType;
extern NXAtom NXFontPboardType;
extern NXAtom NXRulerPboardType;

/*
 * standard Pasteboard names
 */
extern NXAtom NXSelectionPboard;
extern NXAtom NXFontPboard;
extern NXAtom NXRulerPboard;
extern NXAtom NXFindPboard;
```

PopUpList

INHERITS FROM	Menu : Panel : Window : Responder : Object
DECLARED IN	appkit/PopUpList.h

CLASS DESCRIPTION

PopUpList is used to create a pop-up list of items. The list is popped up in response to the action message **popUp:**, usually sent from a Button that acts as a "cover" for the PopUpList. The sender of the **popUp:** message must respond to the messages **title** and **setTitle:**; it can be any subclass of View. If the sender is a Matrix, the **selectedCell** must respond to those messages. In the Interface Builder, a PopUpList and a Button to activate it are available as a single palette item.

A PopUpList can actually be one of two types: pop-up or pull-down. In the Interface Builder, you can select the type by selecting the appropriate icon in the Inspector panel. A pop-up list's button title changes as items are selected from the list; a pull-down list's button title doesn't change.

Accessing the PopUpList's Button is useful if you want to change the title displayed for the list. To access the Button from your code, give it a tag in the Interface Builder's Inspector. Send a **setTitle**: message to the Button to change the title string it displays. If the title you send isn't represented in the PopUpList, it's added at the top of the list the next time the user manipulates the Button.

PopUpList is *not* a control. When you invoke **setAction:** and **setTarget:**, you are setting the action and target of the matrix used to display the list elements. The matrix itself actually sends the action message to the target as items are chosen from the PopUpList.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Responder	id	nextResponder;

Inherited from Window	NXRect	frame;
	id	contentView;
	id	delegate;
	id	firstResponder;
	id	lastLeftHit;
	id	lastRightHit;
	id	counterpart;
	id	fieldEditor;
	int	winEventMask;
	int	windowNum;
	float	backgroundGray;
	struct _wFlags	wFlags;
	struct _wFlags2	wFlags2;
Inherited from Panel	(none)	
Inherited from Menu	id	supermenu;
	id	matrix;
	id	attachedMenu;
	NXPoint	lastLocation;
	id	reserved;
	struct _menuFlags	menuFlags;
Declared in PopUpList	(none)	

METHOD TYPES

Initializing a PopUpList	– init
Setting up the items	– addItem:
	– count
	– indexOfItem:
	– insertItem:at:
	– removeItem:
	– removeItemAt:
Interacting with the Button	– changeButtonTitle:
	– getButtonFrame:
Activating the PopUpList	– popUp:
Returning the user's selection	– selectedItem
Modifying the items	– font
	– setFont:

Target and action	– action – setAction: – setTarget: – target
Resizing the PopUpList	– sizeWindow::

INSTANCE METHODS

action

– (SEL)**action**

Returns the action which will be sent when an item is selected from the list.

See also: – **setAction:**

addItem:

– **addItem:**(const char *)*title*

Adds the item with the name *title* to the PopUpList. The newly added cell is returned. The new item is added to the end of the list.

Note: Popping up a list from a sender whose title is not in the list will cause that title to be added to the list (at the beginning of the list).

See also: – **setTarget:**

changeButtonTitle:

– **changeButtonTitle:**(BOOL)*flag*

If *flag* is YES, then when a selection is made from the list, the title of the selection becomes the title of the Control (usually a Button) which sent the **popUp:** message. If NO, then no such change occurs. YES is the default. Returns **self**.

count

– (unsigned int)**count**

Returns the number of entries in the list.

font

– **font**

Returns the font that is used to draw the items in the PopUpList.

getButtonFrame:

– **getButtonFrame:**(NXRect *)*bframe*

Returns, by reference, the frame for the button which is used to pop this list up.

indexOfItem:

– (int)**indexOfItem:**(const char *)*title*

Returns the index of the item *title*. If *title* is not in the list, returns –1.

init

– **init**

Initializes and returns the receiver, a new instance of PopUpList. This method is the designated initializer for PopUpList. PopUpList does not override the designated initializers for Menu, Panel, or Window. Use only this method to initialize new instances of PopUpList. If you create a subclass of PopUpList that performs its own initialization, you must override this method.

insertItem:at:

– **insertItem:**(const char *)*title* **at:**(unsigned int)*index*

Inserts an item at the specified point in the PopUpList. The **index** starts with item 0 at the top of the list. Returns the newly inserted Cell.

popUp:

– **popUp:***sender*

This is the action message sent by an object, usually a Button, whose target is the PopUpList. The *sender* must be either a subclass of View that responds to the messages **title** and **setTitle:** or a subclass of Matrix whose **selectedCell** responds to **title** and **setTitle:**.

This method works if and only if the Application's **currentEvent** is a mouse down; thus, it should be invoked only as a result of a mouse-down occurring somewhere. When a selection is made in the PopUpList, the Matrix that displays PopUpList's entries sends the action to the target. Returns **self**.

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

removeItem:

– **removeItem:**(const char *)*title*

Removes the item with the name *title* from the list and returns the Cell used to draw the item.

removeItemAt:

– **removeItemAt:**(unsigned int)*index*

Removes the item at the specified *index*. Returns the Cell used to draw the title at that location.

selectedItem

– (const char *)**selectedItem**

Returns the title of the currently selected item. The target of the PopUpList can get the title of the selected item by sending either [[sender **selectedCell**] **title**] or [[sender **window**] **selectedItem**] messages. The former is preferred.

setAction:

– **setAction:**(SEL)*aSelector*

Sets the action sent when an item is selected from the PopUpList. This method invokes the **setAction:** method of the Matrix containing the list of items. Returns **self**.

See also: – **setAction:** (Matrix)

setFont:

– **setFont:***fontId*

Sets the font that is used to draw the PopUpList. Returns **self**.

setTarget:

– **setTarget:***anObject*

Sets the object to which an action will be sent when an item is selected from the list. This method invokes the **setTarget:** method on the Matrix containing the list of items. Returns **self**.

See also: **– setTarget:** (Matrix), **– target**

sizeWindow::

– **sizeWindow:**(NXCoord)*width* **:**(NXCoord)*height*

Never invoke this method directly. This method is overridden from Menu because PopUpList needs to surround itself with a dark gray border, and thus needs to be one pixel wider and taller than a Menu. Returns **self**.

target

– target

Returns the object to which the action will be sent when an item is selected from the list. The default value is **nil**, which causes the action message to be sent down the responder chain.

See also: – **setTarget:**

PrintInfo

INHERITS FROM	Object
DECLARED IN	appkit/PrintInfo.h

CLASS DESCRIPTION

The PrintInfo class contains all information describing a given print job. This includes parameters set in the Page Layout panel, and the Print panel. The units of the paper rectangle and margins are points (72 points equals 1 inch).

The **paperType**, **paperRect**, and **orientation** variables are interrelated. A given paper type has a size, which determines what that paper type's default orientation is (landscape if the width is greater than the height, else portrait). If the user chooses the contrary orientation, the size components in **paperRect** are reversed. These relationships between **paperType**, **paperRect**, and **orientation** must be maintained.

The methods for setting these variables have an **andAdjust:** keyword for a Boolean parameter that can be used to maintain the above relationships. If you pass YES for the parameter, the variables will stayed synchronized. The Page Layout panel performs this maintenance for user actions.

INSTANCE VARIABLES

Inherited from Object

```
Class                  isa;
```

Declared in PrintInfo

```
char                   *paperType;
NXRect                 paperRect;
NXCoord                leftPageMargin;
NXCoord                rightPageMargin;
NXCoord                topPageMargin;
NXCoord                bottomPageMargin;
float                  scalingFactor;
char                   pageOrder;
struct _pInfoFlags {
    unsigned int           orientation:1;
    unsigned int           horizCentered:1;
    unsigned int           vertCentered:1;
    unsigned int           manualFeed:1;
    unsigned int           allPages:1;
    unsigned int           horizPagination:2;
    unsigned int           vertPagination:2;
}                      pInfoFlags;
```

```
int             firstPage;
int             lastPage;
int             currentPage;
int             copies;
char            *outputFile;
DPSContext      context;
char            *printerName;
char            *printerType;
char            *printerHost;
int             resolution;
short           pagesPerSheet;
```

paperType	Type of paper.
paperRect	Rect representing paper's area; origin is always (0,0).
leftPageMargin	Left margin.
rightPageMargin	Right margin.
topPageMargin	Top margin.
bottomPageMargin	Bottom margin.
scalingFactor	Factor to scale image by.
pageOrder	Order of pages in document.
pInfoFlags.orientation	Landscape or portrait mode.
pInfoFlags.horizCentered	True if the image is centered horizontally on the page.
pInfoFlags.vertCentered	True if the image is centered vertically on the page.
pInfoFlags.manualFeed	True if the job requires manual paper feed.
pInfoFlags.allPages	True if all the pages are to be printed.
pInfoFlags.horizPagination	Horizontal pagination.
pInfoFlags.vertPagination	Vertical pagination.
firstPage	First page to print.
lastPage	Last page to print.
currentPage	Current page being printed.

copies	Number of copies to print.
outputFile	File to spool to.
context	Spooling context.
printerName	Name of printer to use.
printerType	Type of that printer.
printerHost	Host machine for that printer. An empty string indicates the local machine.
resolution	Resolution at which to print.
pagesPerSheet	The number of pages per sheet of paper.

METHOD TYPES

Initializing a new PrintInfo instance	– init
Freeing a PrintInfo instance	– free
Defining the printing rectangle	– setMarginLeft:right:top:bottom: – getMarginLeft:right:top:bottom: – setOrientation:andAdjust: – orientation – setPaperRect:andAdjust: – paperRect – setPaperType:andAdjust: – paperType
Setting which pages to print	– setFirstPage: – firstPage – setLastPage: – lastPage – setAllPages: – isAllPages – currentPage
Pagination	– setHorizPagination: – horizPagination – setVertPagination: – vertPagination – setScalingFactor: – scalingFactor

Positioning the image on the page	– setHorizCentered: – isHorizCentered – setVertCentered: – isVertCentered – setPagesPerSheet: – pagesPerSheet
Print job attributes	– setPageOrder: – pageOrder – setManualFeed: – isManualFeed – setCopies: – copies – setResolution: – resolution
Specifying the printer	– setPrinterName: – printerName – setPrinterType: – printerType – setPrinterHost: – printerHost
Spooling	– setOutputFile: – outputFile – setContext: – context
Archiving	– read: – write:

INSTANCE METHODS

context

– (DPSContext)**context**

Returns the Display PostScript context used for printing.

copies

– (int)**copies**

Returns the number of copies of the document that will be printed.

currentPage

– (int)**currentPage**

Returns page number of the page currently being printed. This method is valid only when printing (or faxing) a View. See **setFirstPage:** for the meaning of the number returned.

See also: – **setFirstPage:**, – **printPSCode:** (View)

firstPage

– (int)**firstPage**

Returns the first page that will be printed in this document, assuming **pInfoFlags.allPages** is NO. See **setFirstPage:** for the meaning of the number returned.

See also: – **setFirstPage:**

free

– **free**

Frees all storage used by the PrintInfo object.

getMarginLeft:right:top:bottom:

– **getMarginLeft:**(NXCoord *)*leftMargin*
  right:(NXCoord *)*rightMargin*
  top:(NXCoord *)*topMargin*
  bottom:(NXCoord *)*bottomMargin*

Returns the margins. All margins are in points, in the default coordinate system of the page.

horizPagination

– (int)**horizPagination**

Returns the way in which pagination is done horizontally across the page.

init

– **init**

Initializes the PrintInfo object after memory for it has been allocated by Object's **alloc** or **allocFromZone:** methods. Returns **self**.

isAllPages

– (BOOL)**isAllPages**

Returns whether all the pages of this document are to be printed. If NO, then the pages that are to be printed are given by **firstPage** and **lastPage**.

isHorizCentered

– (BOOL)**isHorizCentered**

Returns whether the default implementation of **placePrintRect:offset:** in the View class centers the image horizontally on the page.

isManualFeed

– (BOOL)**isManualFeed**

Returns whether the pages for this print job will need to be manually fed to the printer.

isVertCentered

– (BOOL)**isVertCentered**

Returns whether the default implementation of **placePrintRect:offset:** in the View class centers the image vertically on the page.

lastPage

– (int)**lastPage**

Returns the last page that will be printed in this document, assuming **allPages** is NO. See **setFirstPage:** for the meaning of the number returned.

See also: – **setFirstPage:**

orientation

– (char)**orientation**

Returns the **orientation** (either NX_PORTRAIT or NX_LANDSCAPE).

outputFile

– (const char *)**outputFile**

Returns the name of the file to which the generated PostScript code is sent. If this field is NULL, output will go to a temporary file.

pageOrder

– (char)**pageOrder**

Returns **pageOrder**.

pagesPerSheet

– (short)**pagesPerSheet**

Returns the number of pages of the document printed per sheet of paper.

paperRect

– (const NXRect *)**paperRect**

Returns a pointer to **paperRect**, which is measures the size of the paper in points.

paperType

– (const char *)**paperType**

Returns the **paperType** of this PrintInfo object. If **paperType** is an unknown type, then an empty string is returned.

printerHost

– (const char *)**printerHost**

Returns the name of the machine where the printer that we will print on resides.

printerName

– (const char *)**printerName**

Returns the name of the printer on which we will print.

printerType

– (const char *)**printerType**

Returns the type of printer on which we will print.

read:

– **read:**(NXTypedStream *)*stream*

Reads the PrintInfo from the typed stream *stream*.

resolution

– (int)**resolution**

Returns the **resolution** at which we will print.

scalingFactor

– (float)**scalingFactor**

Returns **scalingFactor**.

setAllPages:

– **setAllPages:**(BOOL)*flag*

Sets whether all the pages of the document are to be printed (as opposed to a subset given by the **firstPage** and **lastPage** values).

setContext:

– **setContext:**(DPSContext)*aContext*

Sets the DPS **context** we print through. This is normally done by the printing machinery in View.

setCopies:

– **setCopies:**(int)*anInt*

Sets the number of copies of the document that will be printed.

setFirstPage:

– **setFirstPage:**(int)*anInt*

Sets the page number of the first page that will be printed.

Page numbers used by the PrintInfo object should use the same numbering as the pages in the document. For example, if a 10-page document's first page is numbered page 20, then the PrintInfo's first page should be set to 20 and the last page set to 29. This is the same numbering that the user will use to enter specific page ranges in the Print Panel.

setHorizCentered:

– **setHorizCentered:**(BOOL)*flag*

Sets whether the default implementation of **placePrintRect:offset:** in the View class centers the image horizontally on the page.

setHorizPagination:

– **setHorizPagination:**(int)*mode*

Sets the way in which pagination is done horizontally across the page. The value NX_AUTOPAGINATION means the default Application Kit algorithm will be applied to divide the View being printed into pages. The value NX_FITPAGINATION means that the View will be scaled if necessary so that it fits on a single page horizontally. Any scaling applied will also affect the vertical dimension, maintaining a square aspect ratio. The value NX_CLIPPAGINATION means that the View will be clipped horizontally so that there is only one column of pages produced.

setLastPage:

– **setLastPage:**(int)*anInt*

Sets the page number of the last page that will be printed. See **setFirstPage:** for the meaning of the number passed.

See also: – **setFirstPage:**

setManualFeed:

– **setManualFeed:**(BOOL)*flag*

Sets whether the pages for this job will need to be manually fed to the printer.

setMarginLeft:right:top:bottom:

– **setMarginLeft:**(NXCoord)*leftMargin*
  right:(NXCoord)*rightMargin*
  top:(NXCoord)*topMargin*
  bottom:(NXCoord)*bottomMargin*

Sets the margins. All margins are in points, in the default coordinate system of the page.

setOrientation:andAdjust:

– **setOrientation:**(char)*mode* **andAdjust:**(BOOL)*flag*

Sets **orientation**. *mode* should be either NX_PORTRAIT or NX_LANDSCAPE.

If *flag* is NO, then only **orientation** is changed. If *flag* is YES, then **paperRect** is also updated to reflect the new **orientation**.

setOutputFile:

– **setOutputFile:**(const char *)*aString*

Sets the name of the file to which the generated PostScript code is sent. If this field is NULL, output will go to a temporary file.

setPageOrder:

– **setPageOrder:**(char)*mode*

Sets **pageOrder**. *mode* should be one of these constants:

NX_DESCENDINGORDER
NX_SPECIALORDER
NX_ASCENDINGORDER
NX_UNKNOWNORDER

setPagesPerSheet:

– **setPagesPerSheet:**(short)*aShort*

Sets the number of pages of the document printed per sheet of paper. This number is rounded up to a power of two when used by the system.

setPaperRect:andAdjust:

– **setPaperRect:**(const NXRect *)*aRect* **andAdjust:**(BOOL)*flag*

Sets **paperRect**. The origin of the rectangle is always constrained to be (0,0). The origin of *aRect* is ignored. Even though only the size of **paperRect** carries the information, it is stored as a rectangle to facilitate calculations, such as intersecting other objects with this rectangle. Points are the unit of measure.

If *flag* is NO, then only **paperRect** is changed. If *flag* is YES, then **orientation** and **paperType** are updated to reflect the new **paperRect**.

setPaperType:andAdjust:

– **setPaperType:**(const char *)*type* **andAdjust:**(BOOL)*flag*

Sets **paperType** to *type*. If *type* is NULL, **paperType** is set to an empty string.

If *flag* is NO, or if *flag* is YES but *type* is not a recognized paper type, then only **paperType** will be changed. If *flag* is YES and *type* is a known paper type, then **paperRect** and **orientation** are updated to reflect the new type.

setPrinterHost:

– **setPrinterHost:**(const char *)*aString*

Sets the name of the machine where the printer on which we will print resides. If *aString* is an empty string, the host name is set to that of the local machine.

setPrinterName:

– **setPrinterName:**(const char *)*aString*

Sets the name of the printer on which we will print.

setPrinterType:

– **setPrinterType:**(const char *)*aString*

Sets the type of printer on which we will print.

setResolution:

– **setResolution:**(int)*anInt*

Sets the **resolution** at which we will print.

setScalingFactor:

– **setScalingFactor:**(float)*aFloat*

Sets **scalingFactor**.

setVertCentered:

– **setVertCentered:**(BOOL)*flag*

Sets whether the default implementation of **placePrintRect:offset:** in the View class centers the image vertically on the page.

setVertPagination:

– **setVertPagination:**(int)*mode*

Sets the way in which pagination is done vertically across the page. The value NX_AUTOPAGINATION means the default Application Kit algorithm will be applied to divide the View being printed into pages. The value NX_FITPAGINATION means that the View will be scaled if necessary so that it fits on a single page vertically. Any scaling applied will also affect the horizontal dimension, maintaining a square aspect ratio. The value NX_CLIPPAGINATION means that the View will be clipped vertically so that only one row of pages is produced.

vertPagination

– (int)**vertPagination**

Returns the way in which pagination is done vertically across the page.

write:

– **write:**(NXTypedStream *)*stream*

Writes the receiving PrintInfo to the typed stream *stream*.

CONSTANTS AND DEFINED TYPES

```
/* Possible values for the page order */
#define NX_DESCENDINGORDER (-1)    /* descending order of pages */
#define NX_SPECIALORDER      0     /* special order; tells the spooler
                                      to not rearrange pages */
#define NX_ASCENDINGORDER    1     /* ascending order of pages */
#define NX_UNKNOWNORDER      2     /* no page order written out */

/* The orientation of the page */
#define NX_LANDSCAPE         1     /* long side horizontal */
#define NX_PORTRAIT          0     /* long side vertical */

/* Pagination modes */
#define NX_AUTOPAGINATION    0     /* auto pagination */
#define NX_FITPAGINATION     1     /* force image to fit on one page */
#define NX_CLIPPAGINATION    2     /* let image be clipped by page */
```

PrintPanel

INHERITS FROM	Panel : Window : Responder : Object
DECLARED IN	appkit/PrintPanel.h

CLASS DESCRIPTION

PrintPanel is a type of Panel that queries the user for information about the print job, such as which pages and how many copies to print. The PrintPanel contains a Choose button the user can click to display the ChoosePrinter panel and thereby select a printer; see ChoosePrinter's class description for more information.

Printing is typically initiated by the user choosing "Print" in the main menu, which sends a message to a View (or sometimes a Window) to perform its **printPSCode:** method. This method brings up the PrintPanel during the printing process by generating the **shouldRunPrintPanel:** method, which returns YES by default. The PrintPanel is displayed and run using its **runModal** method. This method loads information from the global PrintInfo object, runs the panel using **runModalFor:**, and returns the tag of the button the user clicked to dismiss the panel. See PrintInfo's class specification for details about what information it stores.

You can customize the PrintPanel for your application by adding a View to the panel through **setAccessoryView:**. This View might contain additional controls, for example. If you add a View, you may need to override some of PrintPanel's methods to coordinate any displays or controls you add.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Responder	id	nextResponder;
Inherited from Window	NXRect	frame;
	id	contentView;
	id	delegate;
	id	firstResponder;
	id	lastLeftHit;
	id	lastRightHit;
	id	counterpart;
	id	fieldEditor;
	int	winEventMask;
	int	windowNum;
	float	backgroundGray;
	struct _wFlags	wFlags;
	struct _wFlags2	wFlags2;

Inherited from Panel	(none)	
Declared in PrintPanel	id	appIcon;
	id	pageMode;
	id	firstPage;
	id	lastPage;
	id	copies;
	id	ok;
	id	cancel;
	id	preview;
	id	save;
	id	change;
	id	feed;
	id	resolutionList;
	id	name;
	id	type;
	id	status;
	int	exitTag;
	id	accessoryView;
	id	buttons;

appIcon	The Button containing the application's icon.
pageMode	The Matrix of radio buttons indicating whether to print all pages or a subset.
firstPage	The Form indicating the first page to print.
lastPage	The Form indicating the last page to print.
copies	The TextField indicating how many copies to print.
ok	The Print Button.
cancel	The Cancel Button.
preview	The Preview Button.
save	Save Button.
change	Change Button.
feed	The PopUpList of paper feed options.
resolutionList	The PopUpList of resolution choices.
name	The TextField for the name of the printer.
type	The TextField for the type of printer.

status	The TextField for the printing status.
exitTag	The tag of the button user clicked to exit the panel.
accessoryView	The optional View added by the application.
buttons	The Matrix of PrintPanel buttons.

METHOD TYPES

Creating and freeing a PrintPanel	+ new + newContent:style:backing:buttonMask:defer: – free
Customizing the PrintPanel	– setAccessoryView: – accessoryView
Running the panel	– runModal – pickedButton:
Updating the panel's display	– changePrinter: – pickedAllPages: – textWillChange:
Communicating with the PrintInfo object	– readPrintInfo – writePrintInfo

CLASS METHODS

alloc

Generates an error message. This method cannot be used to create PrintPanel instances; use **new** instead.

See also: + **new**

allocFromZone:

Generates an error message. This method cannot be used to create PrintPanel instances; use **new** instead.

See also: + **new**

new

+ **new**

Creates and returns the PrintPanel. This will return the existing instance of the PrintPanel if one has already been created. To display and run the panel, use the **runModal** method.

See also: – **runModal**

newContent:style:backing:buttonMask:defer:

+ **newContent:**(const NXRect *)*contentRect*
  style:(int)*aStyle*
  backing:(int)*bufferingType*
  buttonMask:(int)*mask*
  defer:(BOOL)*flag*

Used in the instantiation of the PrintPanel. You shouldn't use this method to create the panel; use **new** instead.

See also: + **new**, – **runModal**

INSTANCE METHODS

accessoryView

– **accessoryView**

Returns the View set by **setAccessoryView:**.

See also: – **setAccessoryView:**

changePrinter:

– **changePrinter:***sender*

Brings up the ChoosePrinter Panel to allow the user to select a printer. After the user finishes with that panel, the PrintPanel's display is updated to reflect the newly chosen printer.

free

– **free**

Frees all storage used by the PrintPanel.

pickedAllPages:

– **pickedAllPages:***sender*

Updates the fields for entering page numbers when the user clicks either of the radio buttons indicating whether to print all pages.

pickedButton:

– **pickedButton:***sender*

Ends the current run of this panel by sending the **stopModal** message to the Application object. This method sets the **exitTag** instance variable to the tag of the button that the user clicked to dismiss the panel (either NX_OKTAG, NX_CANCELTAG, NX_PREVIEWTAG, NX_SAVETAG, or NX_FAXTAG).

See also: – **stopModal** (Application)

readPrintInfo

– **readPrintInfo**

Reads the global PrintInfo in Application, setting the initial values of this panel. The number of copies is set at 1, all pages are printed, and automatic feed is chosen.

See also: – **writePrintInfo**

runModal

– (int)**runModal**

Executes the PrintPanel. This method loads the current printing information into the panel from NXApp's global PrintInfo object. It then runs the panel using **runModalFor:**. When the user finishes with the panel, it's still displayed; you must hide the panel when printing is completed. If the user exits the PrintPanel with any button other than cancel, the information in the PrintPanel is written back to the global PrintInfo object. The method returns the tag of the button that the user chose to dismiss the panel (NX_OKTAG, NX_CANCELTAG, NX_SAVETAG, NX_PREVIEWTAG, or NX_FAXTAG). Note that since **runModalFor:** is used, the **pickedButton:** method must use the **stopModal** method to terminate the modal run of this panel.

See also: + **new**

setAccessoryView:

– **setAccessoryView:***aView*

Adds *aView* to the contents of the panel. Applications use this method to add controls to extend the functionality of the panel. The panel is automatically resized to accommodate *aView*, which should be the top View in a view hierarchy. If *aView* is **nil**, then any accessory view in the panel will be removed. **setAccessoryView:** may be performed repeatedly to change the accessory view as needed.

If controls are added, you may need to define your own version of several PrintPanel's methods. For example, you may want to override **pickedAllPages:** to update any fields of information you display. Also, you may need to override **readPrintInfo** and **writePrintInfo** to get information from and write it to the global PrintInfo object.

See also: – **accessoryView:**

textWillChange:

– (BOOL)**textWillChange:***textObject*

Ensures that the correct cell of the page mode matrix is set. Called when the user types in either the first page or last page field of the form.

writePrintInfo

– **writePrintInfo**

Writes the values of the PrintPanel to NXApp's global PrintInfo object.

See also: – **readPrintInfo**

Responder

INHERITS FROM	Object
DECLARED IN	appkit/Responder.h

CLASS DESCRIPTION

Responder is an abstract class that forms the basis of command and event processing in the Application Kit. Most Kit classes inherit from Responder. When a Responder object receives an event or action message that it can't respond to—that it doesn't have a method for—the message is sent to its *next responder.* For a View, the next responder is usually its superview; the content view's next responder is the Window. Each Window, therefore, has its own *responder chain.* Messages are passed up the chain until they reach an object that can respond.

Action messages and keyboard event messages are sent first to the *first responder*, the object that displays the current selection and is expected to handle most user actions within a window. Each Window object has its own first responder. Messages the first responder can't handle work their way up the responder chain.

This class defines the **nextResponder** instance variable and the methods that pass event and action messages along the responder chain.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Declared in Responder	id	nextResponder;

nextResponder	The object that will be sent event messages and action messages that the Responder can't handle.

METHOD TYPES

Managing the next responder	– setNextResponder: – nextResponder
Determining the first responder	– acceptsFirstResponder – becomeFirstResponder – resignFirstResponder
Aiding event processing	– performKeyEquivalent: – tryToPerform:with:

Forwarding event messages	– mouseDown: – rightMouseDown: – mouseDragged: – rightMouseDragged: – mouseUp: – rightMouseUp: – mouseMoved: – mouseEntered: – mouseExited: – keyDown: – keyUp: – flagsChanged: – noResponderFor:
Services menu support	– validRequestorForSendType:andReturnType:
Archiving	– read: – write:

INSTANCE METHODS

acceptsFirstResponder

– (BOOL)**acceptsFirstResponder**

Returns NO to indicate that, by default, Responders don't agree to become the first responder.

Before making any object the first responder, the Application Kit gives it an opportunity to refuse by sending it an **acceptsFirstResponder** message. Objects that can display a selection should override this default to return YES. Objects that respond with this default version of the method will receive mouse event messages, but no others.

See also: **makeFirstResponder:** (Window)

becomeFirstResponder

– becomeFirstResponder

Notifies the receiver that it has just become the first responder for its Window. This default version of the method simply returns **self**. Responder subclasses can implement their own versions to take whatever action may be necessary, such as highlighting the selection.

By returning **self**, the receiver accepts being made the first responder. A Responder can refuse to become the first responder by returning **nil**.

becomeFirstResponder messages are initiated by the Window object (through its **makeFirstResponder:** method) in response to mouse-down events.

See also: – **resignFirstResponder**, – **makeFirstResponder:** (Window)

flagsChanged:

– **flagsChanged:**(NXEvent *)*theEvent*

Passes the **flagsChanged:** event message to the receiver's next responder.

keyDown:

– **keyDown:**(NXEvent *)*theEvent*

Passes the **keyDown:** event message to the receiver's next responder.

keyUp:

– **keyUp:**(NXEvent *)*theEvent*

Passes the **keyUp:** event message to the receiver's next responder.

mouseDown:

– **mouseDown:**(NXEvent *)*theEvent*

Passes the **mouseDown:** event message to the receiver's next responder.

mouseDragged:

– **mouseDragged:**(NXEvent *)*theEvent*

Passes the **mouseDragged:** event message to the receiver's next responder.

mouseEntered:

– **mouseEntered:**(NXEvent *)*theEvent*

Passes the **mouseEntered:** event message to the receiver's next responder.

mouseExited:

– **mouseExited:**(NXEvent *)*theEvent*

Passes the **mouseExited:** event message to the receiver's next responder.

mouseMoved:

– **mouseMoved:**(NXEvent *)*theEvent*

Passes the **mouseMoved:** event message to the receiver's next responder.

mouseUp:

– **mouseUp:**(NXEvent *)*theEvent*

Passes the **mouseUp:** event message to the receiver's next responder.

nextResponder

– **nextResponder**

Returns the receiver's next responder.

See also: – **setNextResponder:**

noResponderFor:

– **noResponderFor:**(const char *)*eventType*

Handles an event message when it's passed to the end of the responder chain and no object can respond. It writes a message to the system log. If the event is a key-down event, it generates a beep.

performKeyEquivalent:

– (BOOL)**performKeyEquivalent:**(NXEvent *)*theEvent*

Returns NO to indicate that, by default, the Responder doesn't have a key equivalent and can't respond to key-down events as keyboard alternatives.

The Responder class implements this method so that any object that inherits from it can be asked to respond to a a **performKeyEquivalent:** message. Subclasses that define objects with key equivalents must implement their own versions of **performKeyEquivalent:**. If the key in *theEvent* matches the receiver's key equivalent, it should respond to the event and return YES.

See also: – **performKeyEquivalent:** (View and Button)

read:

– **read:**(NXTypedStream *)*stream*

Reads the Responder from the typed stream *stream*.

See also: – **write:**

resignFirstResponder

– **resignFirstResponder**

Notifies the receiver that it's no longer the first responder for its window. This default version of the method simply returns **self**. Responder subclasses can implement their own versions to take whatever action may be necessary, such as unhighlighting the selection.

By returning **self**, the receiver accepts the change. By returning **nil**, the receiver refuses to agree to the change, and it remains the first responder.

A **resignFirstResponder** message is sent to the current first responder (through Window's **makeFirstResponder:** method) when another object is about to be made the new first responder.

See also: – **becomeFirstResponder**, – **makeFirstResponder:** (Window)

rightMouseDown:

– **rightMouseDown:**(NXEvent *)*theEvent*

Passes the **rightMouseDown:** event message to the receiver's next responder.

rightMouseDragged:

– **rightMouseDragged:**(NXEvent *)*theEvent*

Passes the **rightMouseDragged:** event message to the receiver's next responder.

rightMouseUp:

– **rightMouseUp:**(NXEvent *)*theEvent*

Passes the **rightMouseUp:** event message to the receiver's next responder.

setNextResponder:

– **setNextResponder:***aResponder*

Makes *aResponder* the receiver's next responder.

See also: – **nextResponder**

tryToPerform:with:

– (BOOL)**tryToPerform:**(SEL)*anAction* **with:***anObject*

Aids in dispatching action messages. This method checks to see whether the receiving object can respond to the method selector specified by *anAction*. If it can, the message is sent with *anObject* as an argument. Typically, *anObject* is the initiator of the action message.

If the receiver can't respond, **tryToPerform:with:** checks to see whether the receiving object's next responder can. It continues to follow next responder links up the responder chain until it finds an object that it can send the action message to, or the chain is exhausted.

Even if the receiver can respond to *anAction* messages, it can "refuse" them by having its implementation of the *anAction* method return **nil**. In this case, the message is passed on to the next responder in the chain.

If successful in finding a receiver that doesn't refuse the message, **tryToPerform:** returns YES. Otherwise, it returns NO.

This method is used (indirectly, through the **sendAction:to:from:** method) to dispatch action messages from Control objects. You'd rarely have reason to use it yourself.

See also: – **sendAction:to:from:** (Application)

validRequestorForSendType:andReturnType:

– **validRequestorForSendType:**(NXAtom)*typeSent*
  andReturnType:(NXAtom)*typeReturned*

Implemented by subclasses to determine what services are available at any given time. In order to keep the Services menu current, the Application object sends **validRequestorForSendType:andReturnType:** messages to the first responder with the send and return types for each service method of every service provider. Thus, a Responder may receive this message many times per event. If the receiving object can place data of type *typeSent* on the pasteboard and receive data of type *typeReturned* back, it should return **self**; otherwise it should return **nil**. The Application object checks the return value to determine whether to enable or disable commands in the Services menu.

Responder's implementation of this method simply forwards the message to the next responder, so by default this method returns **nil**. Like untargetted action messages, **validRequestorForSendType:andReturnType:** messages are passed up the responder chain to the Window, then to the Window's delegate, and finally to the Application object and its delegate, until an object returns **self** rather than **nil**.

typeSent and *typeReturned* are pasteboard types. They're NXAtoms, so you can compare them to the types your application can send and receive by comparing pointers rather than comparing strings. Since this method will be invoked frequently, it must be as efficient as possible.

Either *typeSent* or *typeReturned* may be NULL. If *typeSent* is NULL, the service doesn't require data from the requesting application. If *typeReturned* is NULL, the service doesn't return data to the requesting application.

When the user chooses a menu item for a service, a **writeSelectionToPasteboard:types:** message is sent to the Responder (if *typeSent* was not NULL). The Responder writes the requested data to the pasteboard and a remote message is sent to the service. If the service's *typeReturned* is not NULL, it places return data on the pasteboard, and the Responder receives a **readSelectionFromPasteboard:** message.

The following example demonstrates an implementation of the **validRequestorForSendType:andReturnType:** method for an object that can send and receive ASCII text. Pseudocode is in italics.

```
- validRequestorForSendType:(NXAtom)typeSent
                       andReturnType:(NXAtom)typeReturned
{
    /*
     * First, check to make sure that the types are ones
     * that we can handle.
     */
    if ( (typeSent == NXAsciiPboardType || typeSent == NULL) &&
        (typeReturned == NXAsciiPboardType || typeReturned == NULL) )
    {
        /*
         * If so, return self if we can give the service
         * what it wants and accept what it gives back.
         */
        if ( ((there is a selection) || typeSent == NULL) &&
             ((the text is editable) || typeReturned == NULL) )
        {
            return self;
        }
    }
    /*
     * Otherwise, return the default.
     */
    return [super validRequestorForSendType:typeSent
                                andReturnType:typeReturned];
}
```

See also: – **registerServicesMenuSendTypes:andReturnTypes:** (Application),
– **writeSelectionToPasteboard:types:** (Object Method),
– **readSelectionFromPasteboard:** (Object Method)

write:

– **write:**(NXTypedStream *)*stream*

Writes the receiving Responder to the typed stream *stream*. The next responder is not explicitly written.

See also: – **read:**

SavePanel

INHERITS FROM	Panel : Window : Responder : Object
DECLARED IN	appkit/SavePanel.h

CLASS DESCRIPTION

The SavePanel provides a simple way for an application to query the user for the name of a file to use when saving a document or other data. It allows the application to restrict the filename to have a certain file type, as specified by a filename extension. There is one and only one SavePanel in an application and the **new** method returns a pointer to it.

Whenever the user actually decides on a file name, the message **panelValidateFilename:** will be sent to the SavePanel's delegate (if it responds to that message). The delegate can then determine whether that file name can be used; it returns YES if the file name is okay, or NO if the SavePanel should stay up and wait for the user to type in a different file name. The delegate can also implement a **panel:filterFile:inDirectory:** method to test that both the file name and the directory are valid.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Responder	id	nextResponder;
Inherited from Window	NXRect	frame;
	id	contentView;
	id	delegate;
	id	firstResponder;
	id	lastLeftHit;
	id	lastRightHit;
	id	counterpart;
	id	fieldEditor;
	int	winEventMask;
	int	windowNum;
	float	backgroundGray;
	struct _wFlags	wFlags;
	struct _wFlags2	wFlags2;
Inherited from Panel	(none)	

Declared in SavePanel

```
id                    form;
id                    browser;
id                    okButton;
id                    accessoryView;
id                    separator;
char                  *filename;
char                  *directory;
const char            **filenames;
char                  *requiredType;
struct _spFlags {
    unsigned int          opening:1;
    unsigned int          exitOk:1;
    unsigned int          allowMultiple:1;
    unsigned int          dirty:1;
    unsigned int          invalidateMatrices:1;
    unsigned int          filtered:1;
}                     spFlags;
unsigned short        directorySize;
```

form	Typeable form
browser	The browser
okButton	The OK button
accessoryView	Application-customized area
separator	Line separating icon from rest
filename	The chosen file name
directory	The directory of the chosen file
filenames	The list of chosen files
requiredType	The type of file to save
spFlags.opening	Opening or saving
spFlags.exitOk	Exit status
spFlags.allowMultiple	Whether to allow multiple files
spFlags.dirty	Dirty flag for invisible updates
spFlags.invalidateMatrices	Whether the matrices are valid
spFlags.filtered	Whether types are filtered
directorySize	Current size of directory var

METHOD TYPES

Creating and Freeing a SavePanel	+ newContent:style:backing:buttonMask:defer: – free
Customizing the SavePanel	– setAccessoryView: – accessoryView – setTitle: – setPrompt:
Setting directory and file type	– setDirectory: – setRequiredFileType: – requiredFileType
Running the SavePanel	– runModal – runModalForDirectory:file:
Reading Save information	– directory – filename
Completing a partial filename	– commandKey:
Action methods	– cancel: – ok:
Responding to User Input	– selectText: – textDidGetKeys:isEmpty: – textDidEnd:endChar:
Setting the delegate	– setDelegate: – delegate (Window)

CLASS METHODS

newContent:style:backing:buttonMask:defer:

+ **newContent:**(const NXRect *)*contentRect*
  style:(int)*aStyle*
  backing:(int)*bufferingType*
  buttonMask:(int)*mask*
  defer:(BOOL)*flag*

Creates, if necessary, and returns a new instance of SavePanel. Each application shares just one instance of SavePanel; this method returns the shared instance if it exists. A simpler interface is available via the inherited method **new** which invokes this method with all the appropriate parameters.

INSTANCE METHODS

accessoryView

– **accessoryView**

Returns the view set by **setAccessoryView:**.

See also: **setAccessoryView:**

alloc

Generates an error message. This method cannot be used to create SavePanel instances. Use the **newContent:style:backing:buttonMask:defer:** method instead.

See also: + **newContent:style:backing:buttonMask:defer:**

allocFromZone:

Generates an error message. This method cannot be used to create SavePanel instances. Use the **newContent:style:backing:buttonMask:defer:** method instead.

See also: **newContent:style:backing:buttonMask:defer:**

cancel:

– **cancel:***sender*

This method is the target of the Cancel button in the SavePanel. Returns **self**.

commandKey:

– (BOOL)**commandKey:**(NXEvent *)*theEvent*

This method is used to accept command-key events. If *theEvent* contains a Command-Space, the SavePanel will do file name completion; if it contains a Command-H, the SavePanel jumps to the user's home directory. Other command-key events are ignored. Returns YES

directory

– (const char *)**directory**

Returns the path of the directory that the SavePanel is currently showing.

filename

– (const char *)**filename**

Returns the file name (fully specified) that the SavePanel last accepted. Use **strrchr(**[savepanel **filename**], '/'**)** to get the file name only (no path).

free

– **free**

Frees all storage used by the SavePanel.

ok:

– **ok:***sender*

This method is the target of the OK button in the SavePanel.

requiredFileType

– (const char *)**requiredFileType**

Returns the last type set by **setRequiredFileType:**.

runModal

– (int)**runModal**

Displays the panel and begins its event loop. Returns 1 if successful, 0 otherwise.

runModalForDirectory:file:

– (int)**runModalForDirectory:**(const char *)*path* **file:**(const char *)*filename*

Initializes the panel to the file specified by path and name, then displays it and begins its event loop. Returns 1 if successful, 0 otherwise.

selectText:

– **selectText:***sender*

Advances the current browser selection one line when TAB is pressed (goes back one line when BACKTAB is pressed).

setAccessoryView:

– **setAccessoryView:***aView*

aView should be the top View in a view hierarchy which will be added just above the "OK" and "Cancel" buttons at the bottom of the panel. The panel is automatically resized to accommodate aView. This may be called repeatedly to change the accessory view depending on the situation. If *aView* is **nil**, then any accessory view which is in the panel will be removed.

setDelegate:

– **setDelegate:***anObject*

Makes *anObject* the SavePanel's delegate. Returns **self**.

setDirectory:

– **setDirectory:**(const char *)*path*

Sets the current path in the SavePanel browser. Returns **self**.

setPrompt:

– **setPrompt:**(const char *)*prompt*

Sets the title for the form field in which users type their entries on the panel. This title will appear on all SavePanels (or all OpenPanels if the receiver of this message is an OpenPanel) in your application. "File:" is the default prompt string. Returns **self**.

setRequiredFileType:

– **setRequiredFileType:**(const char *)*type*

Specifies the *type*, a file name extension to be appended to any selected files which do not already have that extension; for example, "nib". *type* should not include the period which begins the extension. Be careful to invoke this method each time the SavePanel is used for another file type within the application. Returns **self**.

setTitle:

– **setTitle:**(const char *)*newTitle*

Sets the title of the SavePanel to *newTitle* and returns **self**. By default, "Save" is the title string. If a SavePanel is adapted to other uses, its title should reflect the user action that brings it to the screen.

textDidEnd:endChar:

– **textDidEnd:***textObject* **endChar:**(unsigned short)*endChar*

Determines whether the key that ended text was Tab or Shift-Tab so that **selectText:** knows whether to move forward or backwards. Returns **self**.

textDidGetKeys:isEmpty:

– **textDidGetKeys:***textObject* **isEmpty:**(BOOL)*flag*

Invoked by the Panel's text to indicate whether there is any text in the Panel. Disables the OK button if there is no text in the Panel.

METHODS IMPLEMENTED BY THE DELEGATE

panel:filterFile:inDirectory:

–(BOOL) **panel:***sender*
 filterFile:(const char *)*filename*
 inDirectory:(const char *)*directory*

Sent to the panel's delegate. The delegate can then determine whether that *filename* can be saved in the *directory*; it returns YES if the *filename* and *directory* are okay, or NO if the SavePanel should stay up and wait for the user to type in a different file name or select another directory.

panelValidateFilenames:

–(BOOL) **panelValidateFilenames:***sender*

Sent to the panel's delegate. The delegate can then determine whether that file name can be used; it returns YES if the file name is okay, or NO if the SavePanel should stay up and wait for the user to type in a different file name.

Scroller

INHERITS FROM	Control : View : Responder : Object
DECLARED IN	appkit/Scroller.h

CLASS DESCRIPTION

The Scroller class defines a Control that's used by a ScrollView object to position a document that's too large to be displayed in its entirety within a View. A Scroller is typically represented on the screen by a bar, a knob, and two scroll buttons, although it may contain only a subset of these. The knob indicates both the position within the document and the amount displayed relative to the size of the document. The bar is the rectangular region that the knob slides within. The scroll buttons allow the user to scroll in small increments by clicking, or in large increments by Alternate-clicking. In discussions of the Scroller class, a small increment is referred to as a "line increment" (even if the Scroller is oriented horizontally), and a large increment is referred to as a "page increment," although a page increment actually advances the document by one windowful. When you create a Scroller, you can specify either a vertical or a horizontal orientation.

As a Control, a Scroller handles mouse events and sends action messages to its target (usually its parent ScrollView) to implement user-controlled scrolling. The Scroller must also respond to messages from a ScrollView to represent changes in document positioning.

Scroller is a public class primarily for programmers who decide not to use a ScrollView but want to present a consistent user interface. Its use is not encouraged except in cases where the porting of an existing application is made more straightforward. In these situations, you initialize a newly created Scroller with **initFrame:**. Then, you use **setTarget:** (Control) to set the object that will receive messages from the Scroller, and you use **setAction:** (Control) to specify the target's method that will be invoked by the Scroller. When your target receives a message from the Scroller, it will probably need to query the Scroller using the **hitPart** and **floatValue** methods to determine what action to take.

The Scroller class has several constants referring to the parts of a Scroller. A scroll button with an up arrow (or left arrow, if the Scroller is oriented horizontally) is known as a "decrement line" button if it receives a normal click, and as a "decrement page" button if it receives an Alternate-click. Similarly, a scroll button with a down or right arrow functions as both an "increment line" button and an "increment page" button. The constants defining the parts of a Scroller are as follows:

Constant	**Refers To**
NX_NOPART	No part of the Scroller
NX_KNOB	The knob
NX_DECPAGE	The button that decrements a page (up, left arrow)
NX_INCPAGE	The button that increments a page (down, right arrow)
NX_DECLINE	The button that decrements a line (up, left arrow)
NX_INCLINE	The button that increments a line (down, right arrow)
NX_KNOBSLOT or NX_JUMP	The bar

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Responder	id	nextResponder;
Inherited from View	NXRect	frame;
	NXRect	bounds;
	id	superview;
	id	subviews;
	id	window;
	struct __vFlags	vFlags;
Inherited from Control	int	tag;
	id	cell;
	struct _conFlags	conFlags;
Declared in Scroller	float	curValue;
	float	perCent;
	int	hitPart;
	id	target;
	SEL	action;
	struct _sFlags{	
	unsigned int	isHoriz:1;
	unsigned int	arrowsLoc:2;
	unsigned int	partsUsable:2;
	}	sFlags;

curValue	The position of the knob, from 0.0 (top or left position) to 1.0.
perCent	The fraction of the bar the knob fills, from 0.0 to 1.0.
hitPart	Which part got the last mouse-down event.
target	The target of the Scroller.

action	The action sent to Scroller's target.
sFlags.isHoriz	True if this is a horizontal Scroller.
sFlags.arrowsLoc	The location of the scroll buttons within the Scroller.
sFlags.partsUsable	The parts of the Scroller that are currently displayed.

METHOD TYPES

Initializing a Scroller	– initFrame:
Laying out the Scroller	– calcRect:forPart: – checkSpaceForParts – setArrowsPosition:
Setting Scroller values	– floatValue – setFloatValue: – setFloatValue::
Resizing the Scroller	– sizeTo::
Displaying	– drawArrow:: – drawKnob – drawParts – drawSelf:: – highlight:
Target and action	– setAction: – action – setTarget: – target
Handling events	– acceptsFirstMouse – hitPart – mouseDown: – testPart: – trackKnob: – trackScrollButtons:
Archiving	– awake – read: – write:

INSTANCE METHODS

acceptsFirstMouse

– (BOOL)**acceptsFirstMouse**

Overrides inherited methods to ensure that the Scroller will receive the mouse-down event that made its window the key window. Returns YES.

action

– (SEL)**action**

Returns the action of the Scroller—in other words, the selector for the method the Scroller will invoke when it receives a mouse-down event.

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

awake

– **awake**

Overrides Object's **awake** method to ensure additional initialization. After a Scroller has been read from an archive file, it will receive this message. You should not invoke this method directly. Returns **self**.

calcRect:forPart:

– (NXRect *)**calcRect:**(NXRect *)*aRect* **forPart:**(int)*partCode*

Calculates the rectangle (in the Scroller's drawing coordinates) that encloses a particular part of the Scroller. This rectangle is returned in *aRect*. *partCode* is NX_DECPAGE, NX_KNOB, NX_INCPAGE, NX_DECLINE, NX_INCLINE, or NX_KNOBSLOT. This method is useful if you override the **drawArrow::** or **drawKnob** method. Returns *aRect* (the pointer you passed it).

See also: – **drawArrow::**, – **drawKnob**

checkSpaceForParts

– **checkSpaceForParts**

Checks to see if there is enough room in the Scroller to display the knob and buttons and sets **sFlags.partsUsable** to one of the following values:

Value	**Meaning**
NX_SCROLLERNOPARTS	Scroller has no usable parts, only the bar.
NX_SCROLLERONLYARROWS	Scroller has only scroll buttons.
NX_SCROLLERALLPARTS	Scroller has all parts.

This method is used by **sizeTo::**; you should not invoke this method yourself. Returns **self**.

See also: – **sizeTo::**

drawArrow::

– **drawArrow:**(BOOL)*upOrLeft* **:**(BOOL)*highlight*

Draws a scroll button. If *upOrLeft* is NO, this method draws the down or right scroll button (NX_INCLINE), depending on whether the Scroller is oriented vertically or horizontally. If *upOrLeft* is YES, this method draws the up or left scroll button (NX_DECLINE). The highlight state is determined by *highlight*. If *highlight* is YES, the button is drawn highlighted, otherwise it's drawn normally. This method is invoked by **drawSelf::** and mouse-down events. It's a public method so that you can override it; you should not invoke it directly. Returns **self**.

See also: – **drawKnob**, – **calcRect:forPart:**

drawKnob

– **drawKnob**

Draws the knob. Don't send this message directly; it's invoked by **drawSelf::** and mouse-down events. Returns **self**.

See also: – **drawArrow::**, – **calcRect:forPart:**

drawParts

– **drawParts**

This method caches images for the various graphic entities composing the Scroller. It's invoked only once by the first of either **initFrame:** or **awake**. You may want to override this method if you alter the look of the Scroller, but you should not invoke it directly. Returns **self**.

drawSelf::

– **drawSelf:**(const NXRect *)*rects* **:**(int)*rectCount*

This method draws the Scroller. It's used by the display methods, and you should not invoke it directly. *rects* is an array of rectangles that need to be covered, with the first one being the union of the subsequent rectangles. *rectCount* is the number of elements in this array. Returns **self.**

See also: – **display:::** (View)

floatValue

– (float)**floatValue**

Returns the position of the knob, a value in range 0.0 to 1.0. A value of 0.0 indicates that the knob is at the top or left position within the bar, depending on the Scroller's orientation.

highlight:

– **highlight:**(BOOL)*flag*

This method highlights or unhighlights the scroll button that the user clicked on. The Scroller invokes this method while tracking the mouse, and you should not invoke it directly. If *flag* is YES, the button is drawn highlighted, otherwise it's drawn normally. Returns **self**.

See also: – **drawArrow::**

hitPart

– (int)**hitPart**

Returns the part of the Scroller that received a mouse-down event. See the Scroller class description for possible values. This method is typically invoked by the ScrollView to determine what action to take when the ScrollView receives an action message from the Scroller.

See also: – **action**

initFrame:

– **initFrame:**(const NXRect *)*frameRect*

Initializes a new Scroller with frame *frameRect*, which cannot be NULL. If *frameRect*'s width is greater than its height, a horizontal Scroller is created; otherwise, a vertical Scroller is created. The Scroller is initially disabled. If the Scroller is a subview of a ScrollView, its width and height are reset automatically by the ScrollView's **tile** method; in this case, the width of vertical Scrollers and the height of horizontal Scrollers are set to NX_SCROLLERWIDTH. This method is the designated initializer for the Scroller class. Returns **self**.

See also: – **setEnabled:** (Control), – **tile** (ScrollView), + **alloc** (Object), + **allocFromZone:** (Object)

mouseDown:

– **mouseDown:**(NXEvent *)*theEvent*

This method acts as a dispatcher when a mouse-down event occurs in the Scroller. It determines what part of the Scroller was clicked, and invokes the appropriate methods (such as **trackKnob:** or **trackScrollButtons:**). You should not invoke this method directly. Returns **self**.

read:

– **read:**(NXTypedStream *)*stream*

Reads the Scroller from the typed stream *stream*, and sets all aspects of its state. Returns **self**.

See also: – **write:**

setAction:

– **setAction:**(SEL)*aSelector*

Sets the action of the Scroller. When the user manipulates the Scroller, the Scroller sends its action message to its target, which (if it's a ScrollView) will then query the Scroller to determine how to respond. Returns **self**.

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

setArrowsPosition:

– **setArrowsPosition:**(int)*where*

Sets the location of the scroll buttons within the Scroller to *where*, or inhibits their display, as follows:

Value	**Meaning**
NX_SCROLLARROWSMAXEND	Buttons at bottom or right
NX_SCROLLARROWSMINEND	Buttons at top or left
NX_SCROLLARROWSNONE	No buttons

Returns **self**.

setFloatValue:

– **setFloatValue:**(float)*aFloat*

Sets the position of the knob to *aFloat*, which is a value between 0.0 and 1.0. This method is the same as **setFloatValue::** except it doesn't change the size of the knob. Returns **self**.

See also: – **setFloatValue::**

setFloatValue::

– **setFloatValue:**(float)*aFloat* **:**(float)*knobProportion*

Sets the position and size of the knob. Sets the position within the bar to *aFloat*, which is a value between 0.0 and 1.0. A value of 0.0 positions and displays the knob at the top or left of the bar, depending on the orientation of the Scroller. The size of the knob is determined by *knobProportion*, which is a value between 0.0 and 1.0. A value of 0.0 sets the knob to a predefined minimum size, and a value of 1.0 makes the knob fill the bar. Returns **self**.

See also: – **setFloatValue:**

setTarget:

– **setTarget:***anObject*

Sets the target of the Scroller. The Scroller's target receives the action message set by **setAction:** when the user manipulates the Scroller. Returns **self**.

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

sizeTo::

– **sizeTo:**(NXCoord)*width* **:**(NXCoord)*height*

Overrides the default View method so the Scroller can check which parts can be drawn. This method is typically invoked by **tile** (ScrollView), which sets the Scroller to a constant width (or height, if the Scroller is horizontal) of NX_SCROLLERWIDTH. Returns **self**.

See also: – **checkSpaceForParts**, – **tile** (ScrollView)

target

– **target**

Returns the Scroller's target.

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

testPart:

– (int)**testPart:**(const NXPoint *)*thePoint*

Returns the part of the Scroller that lies under *thePoint*. See the Scroller class description for possible values.

trackKnob:

– **trackKnob:**(NXEvent *)*theEvent*

Tracks the knob and sends action messages to the Scroller's target. This method is invoked when the Scroller receives a mouse-down event in the knob. You should not invoke this method directly. Returns **self**.

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

trackScrollButtons:

– **trackScrollButtons:**(NXEvent *)*theEvent*

Tracks the scroll buttons and sends action messages to the Scroller's target. This method is invoked when the Scroller receives a mouse-down event in a scroll button. You should not invoke this method directly. Returns **self**.

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

write:

– **write:**(NXTypedStream *)*stream*

Writes the Scroller to the typed stream *stream*, saving all aspects of its state. Returns **self**.

See also: – **read:**

CONSTANTS AND DEFINED TYPES

```
/* Location of scroll buttons within the Scroller */
#define NX_SCROLLARROWSMAXEND   0
#define NX_SCROLLARROWSMINEND   1
#define NX_SCROLLARROWSNONE     2

/* Usable parts in the Scroller */
#define NX_SCROLLERNOPARTS      0
#define NX_SCROLLERONLYARROWS   1
#define NX_SCROLLERALLPARTS     2

/* Part codes for various parts of the Scroller */
#define NX_NOPART               0
#define NX_DECPAGE              1
#define NX_KNOB                 2
#define NX_INCPAGE              3
#define NX_DECLINE              4
#define NX_INCLINE              5
#define NX_KNOBSLOT             6
#define NX_JUMP                 6

#define NX_SCROLLERWIDTH      (18.0)
```

ScrollView

INHERITS FROM	View : Responder : Object
DECLARED IN	appkit/ScrollView.h

CLASS DESCRIPTION

The purpose of the ScrollView class is to allow the user to interact with a document that is too large to be represented in its entirety within a View and must therefore be scrolled. The responsibility of a ScrollView is to coordinate scrolling behavior between Scroller objects and a ClipView object. Thus, the user may drag the knob of a Scroller and the ScrollView will send a message to its ClipView to ensure that the viewed portion of the document reflects the position of the knob. Similarly, the application can change the viewed position within a document and the ScrollView will send a message to the Scrollers advising them of this change.

The ScrollView has at least one subview (a ClipView object), which is called the *content view*. The content view in turn has a subview called the *document view*, which is the view to be scrolled. When a ScrollView is created, it has neither a vertical nor a horizontal scroller, and the content view is sized to fill the ScrollView. If Scrollers are required, the application must send the **setVertScrollerRequired:YES** and **setHorizScrollerRequired:YES** messages to the ScrollView, and the content view is resized to fill the area of the ScrollView not occupied by the Scrollers. These two methods only set flags for the ScrollView; if the flag is YES, the ScrollView will automatically enable and disable the Scroller as required to allow the user to scroll through the entire document. In other words, if the vertical scroller flag is set to YES and the document view grows beyond the vertical bounds of the ClipView, the ScrollView will enable the vertical Scroller.

When a Scroller is required, the application must send the appropriate message to the ScrollView (**setVertScrollerRequired:** or **setHorizScrollerRequired:**). The ScrollView will then create a new Scroller instance, make the Scroller a subview of the ScrollView, and set itself as the Scroller's target. When the ScrollView receives an action message from the Scroller, it queries the Scroller to determine what action to take, and then it sends a message to the content view telling it to scroll itself to the appropriate position. Similarly, when the application modifies the scroll position within the document, it should send a **reflectScroll:** message to the ScrollView, which will then query the content view and set the Scroller(s) accordingly. The **reflectScroll:** message may also cause the ScrollView to enable or disable the Scrollers as required.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Responder	id	nextResponder;
Inherited from View	NXRect	frame;
	NXRect	bounds;
	id	superview;
	id	subviews;
	id	window;
	struct __vFlags	vFlags;
Declared in ScrollView	id	vScroller;
	id	hScroller;
	id	contentView;
	float	pageContext;
	float	lineAmount;

vScroller	The vertical scroller.
hScroller	The horizontal scroller.
contentView	The content view.
pageContext	The amount from the previous page (in the content view's coordinates) remaining in the content view after a page scroll.
lineAmount	The number of units (in the content view's coordinates) to scroll for a line scroll.

METHOD TYPES

Initializing a ScrollView	– initFrame:
Determining component sizes	– getContentSize: – getDocVisibleRect:
Laying out the ScrollView	+ getContentSize:forFrameSize:horizScroller: vertScroller:borderType: + getFrameSize:forContentSize:horizScroller: vertScroller:borderType: – resizeSubviews: – setHorizScrollerRequired: – setVertScrollerRequired: – tile

Managing component Views	– setDocView:
	– docView
	– setHorizScroller:
	– horizScroller
	– setVertScroller:
	– vertScroller
	– reflectScroll:
Modifying graphic attributes	– setBorderType:
	– borderType
	– setBackgroundGray:
	– backgroundGray
	– setBackgroundColor:
	– backgroundColor
Setting scrolling behavior	– setCopyOnScroll:
	– setDisplayOnScroll:
	– setDynamicScrolling:
	– setLineScroll:
	– setPageScroll:
Displaying	– drawSelf::
Managing the cursor	– setDocCursor:
Archiving	– read:
	– write:

CLASS METHODS

getContentSize:forFrameSize:horizScroller:vertScroller:borderType:

+ **getContentSize:**(NXSize *)*cSize*
    forFrameSize:(const NXSize *)*fSize*
    horizScroller:(BOOL)*hFlag*
    vertScroller:(BOOL)*vFlag*
    borderType:(int)*aType*

Calculates the size of a content view for a ScrollView with frame size *fSize*. *hFlag* is YES if the ScrollView has a horizontal scroller, and *vFlag* is YES if it has a vertical scroller. *aType* indicates whether there's a line, a bezel, or no border around the frame of the ScrollView, and is NX_LINE, NX_BEZEL, or NX_NOBORDER. The content view size is placed in the structure specified by *csize*. If the ScrollView object already exists, you can send it a **getContentSize:** message to get the size of its content view. Returns **self**.

See also:
+ getFrameSize:forContentSize:horizScroller:vertScroller:borderType:,
– getContentSize:

getFrameSize:forContentSize:horizScroller:vertScroller:borderType:

+ **getFrameSize:**(NXSize *)*fSize*
  forContentSize:(const NXSize *)*cSize*
  horizScroller:(BOOL)*hFlag*
  vertScroller:(BOOL)*vFlag*
  borderType:(int)*aType*

Calculates the size of the frame required for a ScrollView with a content view size *cSize*. The required frame size is placed in the structure specified by *fSize*. *hFlag* is YES if the ScrollView has a horizontal scroller, and *vFlag* is YES if it has a vertical scroller. *aType* indicates whether there's a line, a bezel, or no border around the frame of the ScrollView, and is NX_LINE, NX_BEZEL, or NX_NOBORDER. Returns **self**.

See also:
+ **getContentSize:forFrameSize:horizScroller:vertScroller:borderType:**,
– **getContentSize:**

INSTANCE METHODS

backgroundColor

– (NXColor)**backgroundColor**

Returns the color of the content view's background. This method simply invokes the content view's **backgroundColor** method.

See also: – **setBackgroundColor:**, – **backgroundGray**,
– **backgroundColor** (ClipView)

backgroundGray

– (float)**backgroundGray**

Returns the gray value of the content view's background, a float from 0.0 (black) to 1.0 (white). This method simply invokes the content view's **backgroundGray** method.

See also: – **setBackgroundGray:**, – **backgroundColor**,
– **backgroundGray** (ClipView)

borderType

– (int)**borderType**

Returns a value representing the type of border surrounding the ScrollView. The possible values for the border type are NX_LINE, NX_BEZEL, and NX_NOBORDER.

See also: – **setBorderType:**

docView

– **docView**

Returns the current document view by sending the ScrollView's content view a **docView** message.

See also: – **setDocView:**, – **docView** (ClipView)

drawSelf::

– **drawSelf:**(const NXRect *)*rects* **:**(int)*rectCount*

This method draws the ScrollView. It does not draw the ScrollView's subviews. *rects* is an array of rectangles that need to be covered, with the first one being the union of the subsequent rectangles. *rectCount* is the number of elements in this array. You may want to override this method if you've subclassed the ScrollView to manage additional subviews and if other separation lines need to be drawn. Returns **self**.

See also: – **borderType**, – **display:::** (View)

getContentSize:

– **getContentSize:**(NXSize *)*theSize*

Places the size of the ScrollView's content view in the structure specified by *theSize*. *theSize* is specified in the coordinates of the ScrollView's superview. Returns **self**.

See also: + **getContentSize:forFrameSize:horizScroller:vertScroller:borderType:**

getDocVisibleRect:

– **getDocVisibleRect:**(NXRect *)*aRect*

Gets the portion of the document view visible within the ScrollView's content view. The content view's bounds rectangle, transformed into the document view's coordinates, is placed in the structure specified by *aRect*. This rectangle represents the portion of the document view's coordinate space that's visible through the ClipView. However, the rectangle doesn't reflect the effects of any clipping that may occur above the ClipView itself. Thus, if the ClipView is itself clipped by one of its superviews, this method returns a different rectangle than the one returned by the **getVisibleRect:** method, because the latter reflects the effects of all clipping by superviews. Returns **self**.

See also: – **getDocVisibleRect:** (ClipView), – **getVisibleRect:** (View)

horizScroller

– **horizScroller**

Returns the horizontal scroller, a Scroller object. This method is provided for the rare case where sending a message directly to the Scroller is desired.

See also: – **vertScroller**

initFrame:

– **initFrame:**(const NXRect *)*frameRect*

Initializes the ScrollView, which must be a newly allocated ScrollView instance. The ScrollView's frame rectangle is made equivalent to that pointed to by *frameRect*, which is expressed in the ScrollView's superview's coordinates. This method installs a ClipView as its content view. Clipping is set to NO by a **setClipping:** message (the ScrollView relies on the content view for clipping), opacity is set to YES by a **setOpaque:** message, and auto-resizing of its subview is set to YES by a **setAutoresizeSubviews:** message. When created, the ScrollView has no Scrollers, and its content view fills its bounds rectangle. This method is the designated initializer for the ScrollView class, and can be used to initialize a ScrollView allocated from your own zone. Returns **self**.

See also: + **alloc** (Object), + **allocFromZone:** (Object),
– **setHorizScrollerRequired:**, – **setVertScrollerRequired:**, – **setLineScroll:**,
– **setPageScroll:**

read:

– **read:**(NXTypedStream *)*stream*

Reads the ScrollView from the typed stream *stream*. This method reads the ScrollView, its scrollers, and its content view, which in turn causes the content view's document view to be read. Returns **self**.

See also: – **write:**

reflectScroll:

– **reflectScroll:***cView*

Determines the new settings for the Scrollers by looking at the relationship between the content view's bounds and the document view's frame, and sends the Scrollers a **setFloatValue::** message. If the appropriate extent of the document view's frame is less than or equal to that of the content view's bounds, the corresponding Scroller is disabled. Returns **self**.

See also: – **setFloatValue::** (Scroller)

resizeSubviews:

– **resizeSubviews:**(const NXSize *)*oldSize*

Overrides View's **resizeSubviews:** to retile the ScrollView. This method is invoked when the ScrollView receives a **sizeTo::** message. Returns **self**.

See also: – **tile**

setBackgroundColor:

– **setBackgroundColor:**(NXColor)*color*

Sets the color of the content view's background. This color is used to paint areas inside the content view that aren't covered by the document view. This method simply invokes the content view's **setBackgroundColor:** method. Returns **self**.

See also: – **backgroundColor**, – **setBackgroundGray:**, – **setBackgroundColor:** (ClipView)

setBackgroundGray:

– **setBackgroundGray:**(float)*value*

Sets the gray value of the content view's background. This gray is used to paint areas inside of the content view that aren't covered by the document view. *value* must be in the range from 0.0 (black) to 1.0 (white). To specify one of the four pure shades of gray, use one of these constants:

Constant	Shade
NX_WHITE	White
NX_LTGRAY	Light gray
NX_DKGRAY	Dark gray
NX_BLACK	Black

This method simply invokes the content view's **setBackgroundGray:** method. Returns **self**.

See also: – **backgroundGray**, – **setBackgroundColor:**,
– **setBackgroundGray:** (ClipView)

setBorderType:

– **setBorderType:**(int)*aType*

Determines the border type of the ScrollView. *aType* must be NX_NOBORDER, NX_LINE, or NX_BEZEL. The default is NX_NOBORDER. Returns **self**.

See also: – **borderType**

setCopyOnScroll:

– **setCopyOnScroll:**(BOOL)*flag*

Determines whether the bits on the screen will be copied when scrolling occurs. If *flag* is YES, scrolling will copy as much of a view's bitmap as possible to scroll the view. If *flag* is NO, the entire content view will always be redrawn to perform a scroll. This should only rarely be changed from the default value (YES). This method simply invokes the content view's **setCopyOnScroll:** method. Returns **self**.

See also: – **setCopyOnScroll:** (ClipView)

setDisplayOnScroll:

– **setDisplayOnScroll:**(BOOL)*flag*

Determines whether the results of scrolling will be immediately displayed. If *flag* is YES, the results of scrolling will be immediately displayed. If *flag* is NO, the ClipView is marked as invalid but is not displayed. The ScrollView may then be updated by sending it a **display** message. This should only rarely be changed from the default value (YES). This method simply invokes the content view's **setDisplayOnScroll:** method. Returns **self**.

See also: – **setDisplayOnScroll:** (ClipView), – **display** (View), – **invalidate** (View)

setDocCursor:

– **setDocCursor:***anObj*

Sets the cursor to be used inside the content view by sending a **setDocCursor:** message to the content view. Returns the old cursor.

See also: – **setDocCursor:** (ClipView)

setDocView:

– **setDocView:***aView*

Attaches the document view to the ScrollView. There is one document view per ScrollView, so if there was already a document view for this ScrollView it is replaced. A ScrollView is initialized without a document view. This method simply invokes the content view's **setDocView:** method. Returns the old document view, or **nil** if there was none.

See also: – **docView**, – **setDocView:** (ClipView)

setDynamicScrolling:

– **setDynamicScrolling:**(BOOL)*flag*

Determines whether dragging a scroller's knob will result in dynamic redisplay of the document. If *flag* is YES, scrolling will occur as the knob is dragged. If *flag* is NO, scrolling will occur only after the knob is released. By default, scrolling occurs as the knob is dragged. Returns **self**.

setHorizScroller:

– **setHorizScroller:***anObject*

Sets the horizontal scroller to an instance of a subclass of Scroller. You will rarely need to invoke this method. This method sets the target of *anObject* to the ScrollView and sets *anObject*'s action to the ScrollView's private method that responds to the Scrollers and invokes the appropriate scrolling behavior. To make the scroller visible, you must have previously sent or must subsequently send a **setHorizScrollerRequired:YES** message to the ScrollView. Returns the old scroller.

See also: – **setVertScroller:**

setHorizScrollerRequired:

– **setHorizScrollerRequired:**(BOOL)*flag*

Adds or removes a horizontal scroller for the ScrollView. If *flag* is YES, the ScrollView creates a new Scroller and resizes its other subviews to make space for the Scroller. If *flag* is NO, the Scroller is removed from the ScrollView and the other subviews are resized to fill the ScrollView. When a ScrollView is created, it does not have a horizontal scroller. Once a Scroller is added, it will be enabled and disabled automatically by the ScrollView. This method retiles and redisplays the ScrollView. Returns **self**.

See also: – **tile**

setLineScroll:

– **setLineScroll:**(float)*value*

Sets the amount to scroll the document view when the ScrollView receives a message to scroll one line. *value* is expressed in the content view's coordinates. Returns **self**.

See also: – **setPageScroll:**

setPageScroll:

– **setPageScroll:**(float)*value*

Sets the amount to scroll the document view when the ScrollView receives a message to scroll one page. *value* is the amount common to the content view before and after the page scroll and is expressed in the content view's coordinates. Therefore, setting *value* to 0.0 implies that the entire content view is replaced when a page scroll occurs. Returns **self**.

See also: – **setLineScroll:**

setVertScroller:

– **setVertScroller:***anObject*

Sets the vertical scroller to an instance of a subclass of Scroller. You will rarely need to invoke this method. This method sets the target of *anObject* to the ScrollView and sets *anObject*'s action to the ScrollView's private method that responds to the Scrollers and invokes the appropriate scrolling behavior. To make the scroller visible, you must have previously sent or must subsequently send a **setHorizScrollerRequired:YES** message to the ScrollView. Returns the old scroller.

See also: – **setHorizScroller:**

setVertScrollerRequired:

– **setVertScrollerRequired:**(BOOL)*flag*

Adds or removes a vertical scroller to the ScrollView. If *flag* is YES, the ScrollView creates a new Scroller and resizes its other subviews to make space for the Scroller. If *flag* is NO, the Scroller is removed from the ScrollView and the other subviews are resized to fill the ScrollView. When a ScrollView is created, it does not have a vertical scroller. Once a Scroller is added, it will be enabled and disabled automatically by the ScrollView. This method retiles and redisplays the ScrollView. Returns **self**.

See also: – **tile**

tile

– tile

Tiles the subviews of the ScrollView. You never send a **tile** message directly, but you may override it if you need to have the ScrollView manage additional views. When **tile** is invoked, it's responsible for sizing each of the subviews of the ScrollView, including the content view. This is accomplished by sending each of its subviews a **setFrame:** message. The width of the vertical scroller and the height of the horizontal scroller (if present) are set to NX_SCROLLERWIDTH. A **tile** message is sent whenever the ScrollView is resized, or a vertical or horizontal scroller is added or removed. The method invoking **tile** should then send a **display** message to the ScrollView. Returns **self**.

See also: – **setVertScrollerRequired:**, – **setHorizScrollerRequired:**, – **resizeSubviews:**

vertScroller

– vertScroller

Returns the vertical scroller, a Scroller object. This method is provided for the rare case where sending a message directly to the scroller is required.

See also: – **horizScroller**

write:

– write:(NXTypedStream *)*stream*

Writes the ScrollView to the typed stream *stream*. This method writes the ScrollView, its scrollers, and its content view, which in turn causes the content view's document view to be written. Returns **self**.

See also: – **read:**

SelectionCell

INHERITS FROM	Cell : Object
DECLARED IN	appkit/SelectionCell.h

CLASS DESCRIPTION

SelectionCell is a subclass of Cell used to implement the visualization of hierarchical lists of names. If the cell is a leaf, it displays its text only; otherwise it also displays a right arrow, similar to the way MenuCell indicates submenus.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Cell	char	*contents;
	id	support;
	struct _cFlags1	cFlags1;
	struct _cFlags2	cFlags2;
Declared in SelectionCell	(none)	

METHOD TYPES

Initializing a new SelectionCell	– init
	– initTextCell:
Querying Component Sizes	– calcCellSize:inRect:
Querying the SelectionCell	– isOpaque
	– setLeaf:
Modifying the SelectionCell	– isLeaf
Displaying	– drawInside:inView:
	– drawSelf:inView:
	– highlight:inView:lit:
Archiving	– awake

INSTANCE METHODS

awake

– awake

Caches the arrow bitmaps, if they aren't already and returns the receiver, a newly unarchived instance of SelectionCell. You don't invoke this method; it is invoked as part of the **read** method used to unarchive objects from typed streams.

calcCellSize:inRect:

– calcCellSize:(NXSize *)*theSize* **inRect:**(const NXRect *)*aRect*

Returns, by reference, the minimum width and height required for displaying the SelectionCell in *aRect*. Leaves enough space for a menu arrow.

drawInside:inView:

– drawInside:(const NXRect *)*cellFrame* **inView:***controlView*

Displays the SelectionCell within *cellFrame* in *controlView*. You never invoke this method directly; it is invoked by the **drawSelf** method of *controlView*. Override this method if you create a subclass of SelectionCell that does its own drawing.

drawSelf:inView:

– drawSelf:(const NXRect *)*cellFrame* **inView:***controlView*

Simply invokes **drawInside:inView:** since the SelectionCell has nothing to draw except its insides. You never invoke this method directly; it is invoked by the **drawSelf** method of *controlView*.

highlight:inView:lit:

– highlight:(const NXRect *)*cellFrame*
        inView:*controlView*
        lit:(BOOL)*flag*

Highlights the cell within *cellFrame* in *controlView* if *flag* is YES, unhighlights it if *flag* is NO. Returns **self**.

init

– init

Initializes and returns the receiver, a new instance of SelectionCell, with the title "ListItem." The new instance is set as a leaf.

See also: **– setLeaf:**

initTextCell:

– **initTextCell:**(const char *)*aString*

Initializes and returns the receiver, a new instance of SelectionCell, with *aString* as its title. The new instance is set as a leaf. This method is the designated initializer for SelectionCell; override this method if you create a subclass of SelectionCell that performs its own initialization.

See also: **– setLeaf:**

isLeaf

– (BOOL)**isLeaf**

Returns YES if the cell is a leaf, NO otherwise. If the cell is a leaf, it displays its text only, otherwise it also displays a right arrow like that MenuCell displays to indicate submenus

See also: **– setLeaf:**

isOpaque

– (BOOL)**isOpaque**

Returns YES since SelectionCells touch all the bits in their frame.

setLeaf:

– **setLeaf:**(BOOL)*flag*

If *flag* is YES, sets the Cell to be a leaf, if NO, sets it to be a branch. Leaf SelectionCells display text only; branch SelectionCells also displays a right arrow like that displayed by MenuCell to indicate submenu entries. Returns **self**.

See also: **– isLeaf:**

Slider

INHERITS FROM	Control : View : Responder : Object
DECLARED IN	appkit/Slider.h

CLASS DESCRIPTION

Sliders are Controls that have a sliding knob that can be moved to represent a value between a minimum and a maximum. The action of the Slider can be sent continuously to the target by invoking **setContinuous:** (YES is the default).

Slider (and an accompanying SliderCell) can be dragged into your application from Interface Builder's Palettes panel.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Responder	id	nextResponder;
Inherited from View	NXRect	frame;
	NXRect	bounds;
	id	superview;
	id	subviews;
	id	window;
	struct __vFlags	vFlags;
Inherited from Control	int	tag;
	id	cell;
	struct _conFlags	conFlags;
Declared in Slider	(none)	

METHOD TYPES

Initializing the Slider Class Objects	+ setCellClass:
Initializing a new Slider instance	– initFrame:
Setting Slider Values	– maxValue
	– minValue
	– setMaxValue:
	– setMinValue:
Enabling the Slider	– setEnabled:

Resizing the Slider	– sizeToFit
Handling Events	– acceptsFirstMouse – mouseDown:

CLASS METHODS

setCellClass:

+ **setCellClass:***classId*

Sets the subclass of SliderCell that's used in implementing all Sliders. The default is SliderCell. *classId* should be the value returned by sending a **class** message to SliderCell or a subclass of SliderCell. Returns the id of the Slider class object.

INSTANCE METHODS

acceptsFirstMouse

– (BOOL)**acceptsFirstMouse**

Returns YES since Sliders always accept first mouse.

initFrame:

– **initFrame:**(const NXRect *)*frameRect*

Initializes and returns the receiver, a new instance of Slider. The Slider will be horizontal if *frameRect* is wider than it is high; otherwise it will be vertical. By default, the Slider is continuous. After initializing the Slider, invoke the **sizeToFit** method to resize the Slider to accommodate its knob. This method is the designated initializer for the Slider class.

maxValue

– (double)**maxValue**

Returns the maximum value of the Slider.

minValue

– (double)**minValue**

Returns the minimum value of the Slider.

mouseDown:

– **mouseDown:**(NXEvent *)*theEvent*

Sends a **trackMouse:inRect:ofView:** message to the Slider's cell. Returns **self**.

setEnabled:

– **setEnabled:**(BOOL)*flag*

If *flag* is YES, enables the Slider; if NO, disables the Slider. Redraws the interior of the Slider if autodisplay is on and the enabled state has changed. Returns **self**.

setMaxValue:

– **setMaxValue:**(double)*aDouble*

Sets the maximum value of the Slider and returns **self**.

setMinValue:

– **setMinValue:**(double)*aDouble*

Sets the minimum value of the Slider and returns **self**.

sizeToFit

– **sizeToFit**

The Slider is sized to fit its cell, and its width is adjusted so that its knob fits exactly in its border. Returns **self**.

SliderCell

INHERITS FROM	ActionCell : Cell : Object
DECLARED IN	appkit/SliderCell.h

CLASS DESCRIPTION

The SliderCell is used to implement the Slider Control as well as to provide Matrices of SliderCells. The **trackRect** is the rectangle inside which tracking occurs–the interior of the bezeled area in which the Slider's knob slides.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Cell	char	*contents;
	id	support;
	struct _cFlags1	cFlags1;
	struct _cFlags2	cFlags2;
Inherited from ActionCell	int	tag;
	id	target;
	SEL	action;
Declared in SliderCell	double	value;
	double	maxValue;
	double	minValue;
	NXRect	trackRect;

value	The current value of the slider
maxValue	The maximum allowable value of the slider
minValue	The minimum allowable value of the slider
trackRect	The interior tracking area

METHOD TYPES

Initializing a new SliderCell	– init
Determining Component Sizes	– calcCellSize:inRect:
	– getKnobRect:flipped:

Setting SliderCell Values	– doubleValue – floatValue – intValue – maxValue – minValue – setDoubleValue: – setFloatValue: – setIntValue: – setMaxValue: – setMinValue: – setStringValue: – stringValue
Modifying Graphic Attributes	– isOpaque
Displaying	– drawBarInside:flipped: – drawInside:inView: – drawKnob – drawKnob: – drawSelf:inView:
Target and Action	– isContinuous – setContinuous:
Tracking the Mouse	– continueTracking:at:inView: + prefersTrackingUntilMouseUp – startTrackingAt:inView: – stopTracking:at:inView:mouseIsUp: – trackMouse:inRect:ofView:
Archiving	– awake – read: – write:

CLASS METHODS

prefersTrackingUntilMouseUp

+ (BOOL)**prefersTrackingUntilMouseUp**

Returns YES to enable a SliderCell instance, after a mouse-down event, to track mouse-dragged and mouse-up events even if they occur outside its frame. This ensures that a SliderCell in a matrix doesn't stop responding to user input (and its neighbor start responding) just because the knob isn't dragged in a perfectly straight line. Override this method to allow a SliderCell to stop tracking if the mouse moves outside its frame after a mouse-down event.

INSTANCE METHODS

awake

– **awake**

Reinitializes the receiver's NXImageReps upon unarchiving.

calcCellSize:inRect:

– **calcCellSize:**(NXSize *)*theSize* **inRect:**(const NXRect *)*aRect*

If the width of *aRect* is greater than its height then the SliderCell will be horizontal in which case *theSize->width* returned will be the same as *aRect->width* and *theSize->height* will be the height of the SliderCell bar. Otherwise, the SliderCell will be vertical, and the height will be the same as *aRect->height* and the width will be the width of the bar. Note that it is usually wrong to invoke **calcCellSize:** without the **inRect:** on a SliderCell.

Override this if you draw a different knob on the SliderCell (or if you draw the SliderCell itself differently). You must also override **getKnobRect:flipped:** and **drawKnob:**.

continueTracking:at:inView:

– (BOOL)**continueTracking:**(const NXPoint *)*lastPoint*
  at:(const NXPoint *)*currentPoint*
  inView:*controlView*

Continues tracking by moving the knob to *currentPoint*. Always returns YES. Invokes **getKnobRect:flipped:** to get the current location of the knob and **drawKnob** to draw the new position. Override this if you want to change the way positioning is done (e.g., if you wanted to add fine positioning with the ALTERNATE key).

doubleValue

– (double)**doubleValue**

Returns the value of the SliderCell.

drawBarInside:flipped:

– **drawBarInside:**(const NXRect *)*cellFrame* **flipped:**(BOOL)*flipped*

Draws the slider bar. Override this method if you want to draw your own slider bar.

See also: – **drawSelf:inView:**

drawInside:inView:

– **drawInside:**(const NXRect *)*cellFrame* **inView:***controlView*

Same as **drawSelf:inView:**, but doesn't draw the bezel.

See also: – **drawSelf:inView:**

drawKnob

– **drawKnob**

Draws the knob. You never override this method; override **drawKnob:** instead.

drawKnob:

– **drawKnob:**(const NXRect*)*knobRect*

Draws the knob in *knobRect*. You must override this method if you want to draw your own knob (as well as **getKnobRect:flipped:** and maybe **calcCellSize:inRect:**).

drawSelf:inView:

– **drawSelf:**(const NXRect *)*cellFrame* **inView:***controlView*

Draws the SliderCell bar and knob. The knob is drawn at a position which reflects the current value of the SliderCell. This **drawSelf:inView:** doesn't invoke **drawInside:inView:**.

This method invokes **calcCellSize:inRect:** and centers the resulting sized rectangle in *cellFrame*, draws the bezel, fills the bar with LTGRAY if the cell is disabled, and 0.5 gray if not, then invokes **drawKnob**.

If, for example, you wanted a SliderCell which could be any size, you simply have **calcCellSize:inRect:** return whatever size you deem appropriate, override **getKnobRect:flipped:** to return the correct rectangle to draw the knob in, and **drawKnob:** so that an appropriate knob is drawn.

floatValue

– (float)**floatValue**

Returns the value of the SliderCell as a **float**.

getKnobRect:flipped:

– **getKnobRect:**(NXRect*)*knobRect* **flipped:**(BOOL)*flipped*

This method must be overridden if you do your own knob (as well as **drawKnob:** and maybe **calcCellSize:inRect:**). It returns the rectangle into which the knob will be drawn based on **value, minValue, maxValue** and trackRect (the interior tracking rectangle of the SliderCell). Remember to take into account the flipping of the target view (in *flipped*) in vertical SliderCells.

init

– **init**

Initializes and returns the receiver, a new instance of SliderCell. The **value** is set to 0.0, the **minValue** is set to 0.0, the **maxValue** is set to 1.0, and the SliderCell is continuous.

This method is the designated initializer for SliderCell; override this method if you create a subclass of SliderCell that performs its own initialization. SliderCell doesn't override the Cell class's designated initializer **initIconCell:**; don't use that method to initialize a SliderCell.

See also: **– setContinuous:**, **– setMaxValue:**, **– setMinValue:**

intValue

– (int)**intValue**

Returns the value of the SliderCell as an int.

isContinuous

– (BOOL)**isContinuous**

Returns YES if action message is sent to the target object continuously as mouse-dragged events occur in the Cell; NO if the action is sent periodically or only on mouse-up events.

isOpaque

– (BOOL)**isOpaque**

Returns YES since all SliderCells are opaque.

maxValue

– (double)**maxValue**

Returns the maximum value of the SliderCell.

See also: **– setMaxValue:**

minValue

– (double)**minValue**

Returns the minimum value of the SliderCell.

See also: **– setMinValue:**

read:

– **read:**(NXTypedStream *)*stream*

Reads the SliderCell from the typed stream *stream*. Returns **self**.

setContinuous:

– **setContinuous:**(BOOL)*flag*

If *flag* is YES, sets the SliderCell so that it sends its action message to its target object continuously as mouse-dragged events occur in it. If NO, then the SliderCell sends its action message to its target object only when a mouse-up event occurs. Returns **self**.

setDoubleValue:

– **setDoubleValue:**(double)*aDouble*

Sets the value of the SliderCell to *aDouble*. Updates the SliderCell knob position to reflect the new value and returns **self**.

setFloatValue:

– **setFloatValue:**(float)*aFloat*

Sets the value of the SliderCell to *aFloat*. Updates the SliderCell knob position to reflect the new value and returns **self**.

setIntValue:

– **setIntValue:**(int)*anInt*

Sets the value of the SliderCell to *anInt*. Updates the SliderCell knob position to reflect the new value and returns **self**.

setMaxValue:

– **setMaxValue:**(double)*aDouble*

Sets the maximum value of the SliderCell to *aDouble*. Returns **self**.

setMinValue:

– **setMinValue:**(double)*aDouble*

Sets the minimum value of the SliderCell to *aDouble*. Returns **self**.

setStringValue:

– **setStringValue:**(const char *)*aString*

Parses *aString* for a floating point value. If a floating point value can be parsed, then the SliderCell value is set and the knob position is updated to reflect the new value; otherwise, does nothing. Returns **self**

startTrackingAt:inView:

– (BOOL)**startTrackingAt:**(const NXPoint *)*startPoint* **inView:***controlView*

Begins a tracking session by moving the knob to *startPoint*. Always returns YES.

stopTracking:at:inView:mouseIsUp:

– **stopTracking:**(const NXPoint *)*lastPoint*
    at:(const NXPoint *)*stopPoint*
    inView:*controlView*
    mouseIsUp:(BOOL)*flag*

Ends tracking by moving the knob to *stopPoint*. Returns **self.**

stringValue

– (const char *)**stringValue**

Returns a pointer to the value of the SliderCell, typecast as a string.

trackMouse:inRect:ofView:

– (BOOL)**trackMouse:**(NXEvent *)*theEvent*
    inRect:(const NXRect *)*cellFrame*
    ofView:*controlView*

Tracks the mouse until it goes up or until it goes outside the *cellFrame*. If *cellFrame* is NULL, then it tracks until the mouse goes up. If the SliderCell is continuous (see Cell's **setContinuous:**), then the action will be continuously sent to the target as the mouse is tracked. If *cellFrame* isn't the same *cellFrame* that was passed to the last **drawSelf:inView:**, then this method doesn't track. Returns **self**.

See also: **– setContinous:**

write:

– **write:**(NXTypedStream *)*stream*

Writes the receiving SliderCell to the typed stream *stream* and returns **self**.

Speaker

INHERITS FROM	Object
DECLARED IN	appkit/Speaker.h

CLASS DESCRIPTION

The Speaker class, with the Listener class, puts an Objective-C interface on Mach messaging. Mach messages are the way that applications (tasks) communicate with each other; they're how remote procedure calls (RPCs) are implemented in the Mach operating system.

A remote message is initiated by sending a Speaker instance the very same Objective-C message you want delivered to the remote application. The Speaker translates the message into the correct Mach message format and dispatches it to the receiving application's port. A Listener in the receiving application reads the message from the port queue and translates in back into an Objective-C message, which it tries to delegate to another object.

If the Speaker expects information back from the Listener, it will wait to receive a reply.

Every application must have at least one Speaker and one Listener, if for no other reason but to communicate with the Workspace Manager. If you don't create a Speaker in start-up code and register it as the application's Speaker (with the **setAppSpeaker:** method), the Application object, when it receives a **run** message, will create one for you.

For a general discussion of the Speaker-Listener interaction, see the Listener class. The descriptions here add Speaker-specific information, but don't repeat any of the basic information presented there. In particular, the discussion here doesn't explain the structure of remote messages or the distinction between input and output argument types.

Sending Remote Messages

Before sending a remote message, it's necessary only to provide variables where output information—information returned to the Speaker by the receiving application—can be returned by reference, and to tell the Speaker which port to send the message to.

The example below shows a typical use of the Speaker class:

```
int     msgDelivered, fileOpened;
id      mySpeaker = [[Speaker alloc] init];
port_t  thePort = NXPortFromName("SomeApp", NULL);
                            /* Gets the public port for SomeApp */

if (thePort != PORT_NULL) {
    [mySpeaker setSendPort:thePort];
                            /* Sets the Speaker to send its
                             * next message to SomeApp's port */
    msgDelivered = [mySpeaker openFile:"/usr/foo" ok:&fileOpened];
                            /* Sends the message, here a message
                             * to open the "/usr/foo" file. */
    if (msgDelivered == 0) {
        if (fileOpened == YES)
            . . .
        else
            . . .
    }
}
. . .
[mySpeaker free];           /* Frees the Speaker
                             * when it's no longer needed. */
port_deallocate(task_self(), thePort);
                            /* Frees the application's
                             * send rights to the port. */
```

The **NXPortFromName()** function returns the port registered with the network name server under the name passed in its first argument. The second argument names the host machine; when it's NULL, as in the example above, the local host is assumed.

To find the port of the Workspace Manager, the constant NX_WORKSPACEREQUEST can be passed as the first argument to **NXPortFromName()**. For example:

```
port_t  workspacePort;
workspacePort = NXPortFromName(NX_WORKSPACEREQUEST, NULL);
```

A Speaker can be dedicated to sending remote messages to a single application, in which case its destination port may need to be set only once. Or a single Speaker can be used to send messages to any number of applications, simply by resetting its port.

It's important to reset the destination port of the Speaker registered as the **appSpeaker** before each remote message. The Application Kit uses the **appSpeaker** to keep in contact with the Workspace Manager and so may reset its port behind your application's back.

Return Values

Each method that initiates a remote message returns an **int** that indicates whether the message was successfully transmitted or not.

- If the message couldn't be delivered to the receiving application, the return value will be one of the Mach error codes defined in the **message.h** header file in **/usr/include/sys**.
- If the message was delivered, but the Listener didn't recognize it or couldn't delegate it to a responsible object, the return value is −1.
- If the message was successfully delivered, recognized, and delegated, 0 is returned.

A Mach error code is also returned if the Speaker times out while waiting for a return message.

Copying Output Data

The validity of all output arguments is guaranteed until the next remote message is sent. Then the memory allocated for a character string or a byte array will be freed automatically. If you want to save an output string or an array, you must copy it. When the amount of data is large, you can use the **NXCopyOutputData**() function to take advantage of the out-of-line data feature of Mach messaging. This function is passed the index of the output argument to be copied (the combination of a pointer and an integer for a byte array counts as a single argument) and returns a pointer to an area obtained through the **vm_allocate**() function. This pointer must be freed with **vm_deallocate**(), rather than **free**(). Note that the size of the area allocated is rounded up to the next page boundary, and so will be at least one page. Consequently, it is more efficient to **malloc**() and copy amounts up to about half the page size.

Note: The application is responsible for deallocating all ports received when they're no longer needed.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Declared in Speaker	port_t	sendPort;
	port_t	replyPort;
	int	sendTimeout;
	int	replyTimeout;
	id	delegate;

sendPort	The port to which the Speaker sends remote messages.
replyPort	The port where the Speaker receives return messages from the Listener of the remote application.
sendTimeout	How long the Speaker will wait for a remote message to be delivered at the port of the receiving application.
replyTimeout	How long the Speaker will wait, after a remote message is delivered, to receive a return message from the other application.
delegate	The Speaker's delegate, which is generally unused.

METHOD TYPES

Initializing a new Speaker instance	– init
Freeing a Speaker	– free
Setting up a Speaker	– setSendTimeout: – sendTimeout – setReplyTimeout: – replyTimeout
Managing the ports	– setSendPort: – sendPort – setReplyPort: – replyPort
Standard remote methods	– openFile:ok: – openTempFile:ok: – launchProgram:ok: – powerOffIn:andSave: – extendPowerOffBy:actual: – unmounting:ok:

Handing off an icon	– iconEntered:at::iconWindow:iconX:iconY: iconWidth:iconHeight:pathList: – iconMovedTo:: – iconReleasedAt::ok: – iconExitedAt:: – registerWindow:toPort: – unregisterWindow:
Providing for program control	– msgCalc: – msgCopyAsType:ok: – msgCutAsType:ok: – msgDirectory:ok: – msgFile:ok: – msgPaste: – msgPosition:posType:ok: – msgPrint:ok: – msgQuit: – msgSelection:length:asType:ok: – msgSetPosition:posType:andSelect:ok: – msgVersion:ok:
Getting file information	– getFileIconFor:TIFF:TIFFLength:ok: – getFileInfoFor:app:type:ilk:ok:
Sending remote messages	– performRemoteMethod: – performRemoteMethod:with:length: – selectorRPC:paramTypes:... – sendOpenFileMsg:ok:andDeactivateSelf: – sendOpenTempFileMsg:ok:andDeactivateSelf:
Assigning a delegate	– setDelegate: – delegate
Archiving	– read: – write:

INSTANCE METHODS

delegate

– delegate

Returns the Speaker's delegate.

See also: **– setDelegate:**

extendPowerOffBy:actual:

– (int)**extendPowerOffBy:**(int)*requestedMs* **actual:**(int *)*actualMs*

Sends a remote message requesting more time before the power goes off or the user logs out. This message should be directed to the Workspace Manager. It's sent in response to a **powerOffIn:andSave:** message that doesn't give the application enough time to prepare for the impending shutdown.

requestedMs is how many additional milliseconds are needed, beyond the number given in the **powerOffIn:andSave:** message. The actual number of additional milliseconds that are granted will be returned by reference in the integer referred to by *actualMs*.

See also: **– powerOffIn:andSave:** (Listener and Application),
– app:powerOffIn:andSave: (Application delegate)

free

– free

Frees the memory occupied by the Speaker object, but does not deallocate its ports.

getFileIconFor:TIFF:TIFFLength:ok:

– (int)**getFileIconFor:**(char *)*fullPath*
　　TIFF:(char **)*tiffData*
　　TIFFLength:(int *)*length*
　　ok:(int *)*flag*

Sends a remote message requesting the icon for the *fullPath* file. This request should be directed to the Workspace Manager.

fullPath is a string containing the complete path for a single file. *tiffData* is the address of a pointer that will be set to point to a byte array containing the icon image. The image is provided as TIFF (Tag Image File Format) data. The number of bytes in the *tiffData* array are returned by reference in the integer referred to by *length*.

flag is the address of an integer that will be set to YES if the Workspace Manager provides the icon, and to NO if it doesn't. Here's an example of a method the takes a pathname and returns an NXImage object containing the file's icon:

```
- workspaceImage:(char *)fullPath
{
    int ok, length;
    char *tiffData;
    NXStream *imageStream;
    id theIcon, mySpeaker = [NXApp appSpeaker];

    [mySpeaker setSendPort:
          NXPortFromName(NX_WORKSPACEREQUEST,NULL)];
    [mySpeaker getFileIconFor:fullPath TIFF:&tiffData
          TIFFLength:&length ok:&ok];

    if (!ok) return nil;

    imageStream = NXOpenMemory(tiffData, length, NX_READONLY);
    if (!imageStream) return nil;

    theIcon = [[NXImage alloc] initFromStream:imageStream];
    NXClose(imageStream);

    return theIcon;
}
```

You cannot use **getFileIconFor:...** from within an implementation of the **iconEntered:at:...** Listener method, as the Workspace will be blocked waiting for **iconEntered:at:...** to return. See the documentation for the **iconEntered:at:...**Listener method for information on copying the image of an icon that gets dragged into a window.

See also: – **getFileInfoFor:app:type:ilk:ok:**, – **iconEntered:at:...** (Listener), – **iconReleasedAt::ok:** (Listener)

getFileInfoFor:app:type:ilk:ok:

– (int)**getFileInfoFor:**(char *)*fullPath*
  app:(char **)*appName*
  type:(char **)*aType*
  ilk:(int *)*anIlk*
  ok:(int *)*flag*

Sends a remote message asking for information about the *fullPath* file. This message should be sent to the Workspace Manager, which implements a method that can provide the requested information.

appName is the address of a character pointer; the pointer will be set to point to the name of the application that the Workspace Manager would call upon to open the *fullPath* file.

aType is the address of a pointer that will be set to point to the file type. The type is typically the file name extension—"wn" for WriteNow files and "score" for music files in the ScoreFile language, for example.

anIlk is the address of an integer that will be set to one of the following constants:

NX_ISODMOUNT	*fullPath* is where a file system on an optical disk is mounted.
NX_ISSCSIMOUNT	*fullPath* is where a file system on a hard disk is mounted.
NX_ISNETMOUNT	*fullPath* is where a file system accessed over the network is mounted.
NX_ISDIRECTORY	*fullPath* is a directory, but not one where a file system is mounted and not a file package.
NX_ISAPPLICATION	*fullPath* is an executable file or a ".app" file package for an executable file.
NX_ISFILE	*fullPath* is a file or a file package (not one of the above).

The last argument, *flag*, is the address of an integer that will be set to YES if the Workspace Manager provides the information requested by the three other arguments, and to NO if it doesn't.

To get the icon for *fullPath*, use **getFileIconFor:TIFF:TIFFLength:ok:**.

See also: – **getFileIconFor:TIFF:TIFFLength:ok:**

iconEntered:at::iconWindow:iconX:iconY:iconWidth:iconHeight:pathList:

– (int)**iconEntered:**(int)*windowNum*
  at:(double)*x*
  :(double)*y*
  iconWindow:(int)*iconWindowNum*
  iconX:(double)*iconX*
  iconY:(double)*iconY*
  iconWidth:(double)*iconWidth*
  iconHeight:(double)*iconHeight*
  pathList:(const char *)*pathList*

Sends a remote message notifying another application that the user has dragged an icon into one of its windows. This notification is sent by the Workspace Manager; see the Listener class for information on how to receive

iconEntered:at::iconWindow:iconX:iconY:iconWidth:iconHeight:pathList: messages.

See also: – **registerWindow:toPort:**

iconExitedAt::

– (int)**iconExitedAt:**(double)*x* **:**(double)*y*

Sends a remote message notifying the receiving application that the user dragged an icon out of one its registered windows. This notification is sent by the Workspace Manager; see the Listener class for information on receiving **iconExitedAt::** messages.

See also: – **registerWindow:toPort:**, **iconExitedAt::** (Listener)

iconMovedTo::

– (int)**iconMovedTo:**(double)*x* **:**(double)*y*

Sends a remote message notifying another application that the user dragged an icon within one of its registered windows, to (*x*, *y*) in the screen coordinate system. This notification is sent by the Workspace Manager; see the Listener class for information on receiving **iconMovedTo::** messages.

See also: – **registerWindow:toPort:**, **iconMovedTo::** (Listener)

iconReleasedAt::ok:

– (int)**iconReleasedAt:**(double)*x*
    :(double)*y*
    ok:(int *)*flag*

Sends a remote message notifying another application that the user has dragged an icon to one of its registered windows and released it there, at (*x*, *y*) in screen coordinates. This notification is sent by the Workspace Manager; see the Listener class for information on receiving **iconReleasedAt::ok:** messages.

See also: – **registerWindow:toPort:**, **iconReleasedAt::ok:** (Listener)

init

– **init**

Initializes the Speaker immediately after memory for it has been allocated by Object's **alloc** or **allocFromZone:** methods. The new object's **sendTimeout** and **replyTimeout** are both set to 30,000 milliseconds, its **sendPort** and **replyPort** are both PORT_NULL, and its delegate is **nil**. Returns **self**.

launchProgram:ok:

– (int)**launchProgram:**(const char *)*name* **ok:**(int *)*flag*

Sends a remote message requesting the receiver to launch the *name* application. This message is sent only to the Workspace Manager, the application responsible for executing programs that run in the workspace. *name* is the ordinary name of the application to be launched—for example, "Edit" or "Webster". *flag* points to an integer that will be set to YES if the program is executed, and to NO if it's not.

The Application Kit initiates **launchProgram:ok:** messages when it needs a running application to send another message. For example, the **NXPortFromName()** function uses this method to launch the application you name if it's not already running.

See also: – **openFile:ok:** (Application)

msgCalc:

– (int)**msgCalc:**(int *)*flag*

Sends a remote message asking the receiving application to perform any calculations necessary to update its current window. *flag* points to an integer that will be set to YES if the calculations will be performed, and to NO if they won't.

msgCopyAsType:ok:

– (int)**msgCopyAsType:**(const char *)*aType* **ok:**(int *)*flag*

Sends a remote message asking the receiving application to copy its current selection to the pasteboard as *aType* data. *flag* is the address of an integer that will be set to YES if the selection is copied, and to NO if it isn't.

msgCutAsType:ok:

– (int)**msgCutAsType:**(const char *)*aType* **ok:**(int *)*flag*

Sends a remote message requesting the receiving application to delete the current selection and put it in the pasteboard as *aType* data. *flag* points to an integer that will be set to YES if the request is carried out, and to NO if it isn't.

msgDirectory:ok:

– (int)**msgDirectory:**(char *const *)*fullPath* **ok:**(int *)*flag*

Sends a remote message asking the receiving application for its current directory. See the Listener class for information on the two arguments.

See also: – **msgDirectory:ok:** (Listener)

msgFile:ok:

– (int)**msgFile:**(char *const *)*fullPath* **ok:**(int *)*flag*

Sends a remote message asking the receiving application for its current document (the file displayed in the main window). See the Listener class for information on the two arguments.

See also: – **msgFile:ok:** (Listener)

msgPaste:

– (int)**msgPaste:**(int *)*flag*

Sends a remote message asking the receiving application to replace its current selection with the contents of the pasteboard, just as if the user had chosen the Paste command in the Edit menu. *flag* is the address of an integer that will be set to YES if the receiving application will carry out the request, and to NO if it won't.

msgPosition:posType:ok:

– (int)**msgPosition:**(char *const *)*aString*
  posType:(int *)*anInt*
  ok:(int *)*flag*

Sends a remote message asking the receiving application for information about its current selection. See the Listener class for information on the three arguments.

See also: – **msgPosition:posType:ok:** (Listener)

msgPrint:ok:

– (int)**msgPrint:**(const char *)*fullPath* **ok:**(int *)*flag*

Sends a remote message asking the receiving application to print the *fullPath* file, then close it. *flag* points to an integer that will be set to YES if the file will be printed, and to NO if it won't.

msgQuit:

– (int)**msgQuit:**(int *)*flag*

Sends a remote message requesting the receiving application to quit. *flag* points to an integer that will be set to YES if the receiving application quits, and to NO if it doesn't.

msgSelection:length:asType:ok:

– (int)**msgSelection:**(char *const *)*bytes*
length:(int *)*numBytes*
asType:(const char *)*aType*
ok:(int *)*flag*

Sends a remote message asking the receiving application to provide its current selection as *aType* data. See the Listener class for information on the four arguments.

See also: – **msgSelection:length:asType:ok:** (Listener)

msgSetPosition:posType:andSelect:ok:

– (int)**msgSetPosition:**(const char *)*aString*
posType:(int)*anInt*
andSelect:(int)*sflag*
ok:(int *)*flag*

Sends a remote message asking the receiving application to scroll its current document (the one displayed in the main window) so that the portion represented by *aString* is visible. See the Listener class for information on permitted argument values.

See also: – **msgSetPosition:posType:andSelect:ok:** (Listener)

msgVersion:ok:

– (int)**msgVersion:**(char *const *)*aString* **ok:**(int *)*flag*

Sends a remote message asking the receiving application for its current version. See the Listener class for information on the arguments.

See also: – **msgVersion:ok:** (Listener)

openFile:ok:

– (int)**openFile:**(const char *)*fullPath* **ok:**(int *)*flag*

Sends a remote message requesting another application to open the *fullPath* file. Before the message is sent, the sending application is deactivated to allow the application that will open the file to become the active application.

If the Workspace Manager is sent this message, it will find an appropriate application to open the file based on the file name extension. It will launch that application if necessary.

flag is the address of an integer that the receiving application will set to YES if it opens the file, and to NO if it doesn't.

See also: – **openFile:ok:** (Application)

openTempFile:ok:

– (int)**openTempFile:**(const char *)*fullPath* **ok:**(int *)*flag*

Sends a remote message requesting another application to open a temporary file. The file is specified by an absolute pathname, *fullPath*. Before the message is sent, the sending application is deactivated to allow the application that will open the file to become the active application.

Using this method instead of **openFile:ok:** lets the receiving application know that it should delete the file when it no longer needs it.

See also: – **openTempFile:ok:** (Application)

performRemoteMethod:

– (int)**performRemoteMethod:**(const char *)*methodName*

Sends a remote message to perform the *methodName* method. The method must be one that takes no arguments. **performRemoteMethod:** is analogous to Object's **perform:** method in that it permits you to send an arbitrary message.

This method has the same return values as other methods that send remote messages: 0 on success, a Mach error code if the message couldn't be delivered, and –1 if it was delivered but wasn't understood or couldn't be delegated.

See also: – **selectorRPC:paramTypes:**

performRemoteMethod:with:length:

– (int)**performRemoteMethod:**(const char *)*methodName*
    with:(const char *)*data*
    length:(int)*numBytes*

Sends a remote message to perform the *methodName* method and passes it the *data* byte array containing *numBytes* of data. This method is similar to Object's **perform:with:** method in that it permits you to send an arbitrary message with one argument.

performRemoteMethod:with:length: has the same return values as other methods that send remote messages: 0 on success, a Mach error code if the message couldn't be delivered, and –1 if it was delivered but wasn't understood or couldn't be delegated.

See also: – **selectorRPC:paramTypes:**

powerOffIn:andSave:

– (int)**powerOffIn:**(int)*ms* **andSave:**(int)*aFlag*

Sends a remote message that the power is about to go off, or that the user is about to log out, in *ms* milliseconds. The Workspace Manager is the application that initiates this message, broadcasting it to all running applications. See the Listener and Application classes for information on how to respond to **powerOffIn:andSave:** messages.

See also: – **powerOffIn:andSave:** (Listener and Application)

read:

– **read:**(NXTypedStream *)*stream*

Reads the Speaker from the typed stream *stream*. The Speaker's **sendPort** and **replyPort** instance variables will both be PORT_NULL.

See also: – **write**

registerWindow:toPort:

– (int)**registerWindow:**(int)*windowNum* **toPort:**(port_t)*aPort*

Sends a remote message registering *windowNum*, so that the application will be notified when the user drags an icon over the window. This message should be sent to the Workspace Manager, which displays the file icons that users can drag to other windows. A window must be registered for it to accept icons dragged from the Workspace Manager and other applications.

Once an window is registered, the Workspace Manager will dispatch messages to the application whenever the user drags an icon into, out of, or within the window. The Workspace Manager will also notify the application (with a **iconReleasedAt::ok:** message) when the user drops the icon in the window. The application can then either accept the icon, or reject it and have the Workspace Manager animate it back to its source window.

windowNum is the global window number of the window that accepts icons. The global window number is the Window Server's unique identifier for the window; it can be obtained from the Window object as follows:

```
unsigned int  global;
NXConvertWinNumToGlobal([myWindow windowNum], &global);
```

aPort is the port where the application wants to receive subsequent notification messages from the Workspace Manager.

See also: – **unregisterWindow:**, – **iconEntered:at:...** (Listener),
– **dragFile:fromRect:slideBack:event:** (View),

replyPort

– (port_t)**replyPort**

Returns the port where the Speaker expects to receive return messages. The Speaker caches this port as its **replyPort** instance variable. If this method returns PORT_NULL, the default, the Speaker will use the port returned by Application's **replyPort** method.

See also: – **replyPort** (Application), – **setReplyPort:**

replyTimeout

– (int)**replyTimeout**

Returns how many milliseconds the Speaker will wait, after delivering a remote message to another application, for a return message to arrive back from the other application.

See also: – **setReplyTimeout:**

selectorRPC:paramTypes:

– (int)**selectorRPC:**(const char *)*methodName*
  paramTypes:(char *)*params,*
  ...

Sends a remote message to perform the *methodName* method with an arbitrary number of arguments. This is the general routine for sending remote messages and is used by most of the more specific Speaker methods. For example, a **getFileInfoFor:app:type:ilk:ok:** message could be sent as follows:

```
int      msgDelivered, infoProvided, theIlk;
char    *theApp, *theExtension;

msgDelivered =
    [mySpeaker selectorRPC:"getFileInfoFor:app:type:ilk:ok:"
                     paramTypes:"cCCII","/usr/foo",
                     &theApp, &theExtension,
                     &theIlk, &infoProvided];
```

params is a character string, "cCCII" in the example above, that describes the arguments to the method. Each argument is represented by a single character that encodes its type. (A single character, "b" or "B", represents the two Objective-C arguments of a byte array.) See the Listener class for an explanation of these codes.

The actual arguments that will be passed to *methodName* are listed after *params*.

This method has the same return values as other methods that send remote messages: 0 on success, a Mach error code if the message couldn't be delivered, and –1 if it was delivered but wasn't understood or couldn't be delegated.

sendOpenFileMsg:ok:andDeactivateSelf:

– (int)**sendOpenFileMsg:**(const char *)*fullPath*
  ok:(int *)*flag*
  andDeactivateSelf:(BOOL)*deactivateFirst*

Initiates an **openFile:ok:** remote message, which could also be initiated by sending an **openFile:ok:** message directly to the Speaker. However, when a Speaker receives an **openFile:ok:** message, it first deactivates the application in order to allow the receiving application to become active when it opens the file.

In contrast, this way of sending an **openFile:ok:** remote message gives the sending application control over whether it will deactivate before dispatching the message. If *deactivateFirst* is YES, this method works just like **openFile:ok:**. If *deactivateFirst* is NO, the sending application will remain the active application.

See also: – **openFile:ok:**

sendOpenTempFileMsg:ok:andDeactivateSelf:

– (int)**sendOpenTempFileMsg:**(const char *)*fullPath*
  ok:(int *)*flag*
  andDeactivateSelf:(BOOL)*deactivateFirst*

Initiates an **openTempFile:ok:** remote message, which could also be initiated by sending an **openTempFile:ok:** message directly to the Speaker. However, when a Speaker receives an **openTempFile:ok:** message, it first deactivates the application in order to allow the receiving application to become active when it opens the file.

In contrast, this way of sending an **openTempFile:ok:** remote message gives the sending application control over whether it will deactivate before dispatching the message. If *deactivateFirst* is YES, this method works just like **openTempFile:ok:**. If *deactivateFirst* is NO, the sending application will remain the active application.

See also: – **openTempFile:ok:**

sendPort

– (port_t)**sendPort**

Returns the port the Speaker will send remote messages to. The Speaker caches this port as its **sendPort** instance variable.

See also: – **setSendPort:**

sendTimeout

– (int)**sendTimeout**

Returns how many milliseconds the Speaker will wait for its remote message to be delivered to the port of the receiving application. The Speaker caches this value as its **sendTimeout** instance variable. If it's 0, there's no time limit.

See also: – **setSendTimeout:**

setDelegate:

– **setDelegate:***anObject*

Makes *anObject* the Speaker's delegate. The default delegate is **nil**. But before processing the first event, Application's **run** method makes the Application object, NXApp, the delegate of the Speaker registered as the **appSpeaker**. If there is no **appSpeaker**, the **run** method creates one, registers it, and sets its delegate to be NXApp.

Unlike a Listener, a Speaker doesn't expect anything from its delegate.

See also: – **delegate**, – **setAppSpeaker:** (Application)

setReplyPort:

– **setReplyPort:**(port_t)*aPort*

Makes *aPort* the port where the Speaker receives return messages. If the Speaker sends a remote message with output arguments, it will supply the receiving application with send rights to this port, then wait for a return message containing the output data it requested.

If *aPort* is PORT_NULL, the Speaker will use a port supplied by the Application object in response to a **replyPort** message. Since return messages are read from the port as they arrive (synchronously), a number of different Speakers can share the same port.

At start-up, before the **run** method gets the application's first event, it sets the port of the Speaker registered as the **appSpeaker** to the port returned by Application's **replyPort** method.

See also: – **replyPort**, – **replyPort** (Application)

setReplyTimeout:

– **setReplyTimeout:**(int)*ms*

Sets, to *ms* milliseconds, how long the Speaker will wait to receive a reply from the application it sent a remote message. The Speaker expects a reply when the remote message it sends contains output arguments for information to be supplied by the receiving application. If *ms* is 0, there will be no time limit; the Speaker will wait until a return message is received or there's a transmission error. The default is 30,000 milliseconds.

See also: – **replyTimeout**

setSendPort:

– **setSendPort:**(port_t)*aPort*

Makes *aPort* the port that the Speaker will send remote messages to. The default is PORT_NULL. A single Speaker can send remote messages to a variety of applications simply by setting a different port before each message.

The **NXPortFromName()** function can be used to find the public port of another application, as explained in the class description above.

See also: – **sendPort**

setSendTimeout:

– **setSendTimeout:**(int)*ms*

Sets, to *ms* milliseconds, how long the Speaker will persist in attempting to deliver a message to the port of the receiving application. If *ms* is 0, there will be no time limit; the Speaker will wait until the message is successfully delivered or there's a transmission error. The default is 30,000 milliseconds.

See also: – **sendTimeout**

unmounting:ok:

– (int)**unmounting:**(const char *)*fullPath* **ok:**(int *)*flag*

Sends a remote message that a disk is about to be unmounted. When the user requests it to unmount a disk, the Workspace Manager sends **unmounting:ok:** messages to every running application. Other applications use the Listener version of the method to receive the Workspace Manager's message.

See also: – **unmounting:ok:** (Listener and Application)

unregisterWindow:

– (int)**unregisterWindow:**(int)*windowNum*

Sends a remote message cancelling the registration of *windowNum* as a window that accepts dragged icons. This message should be sent to the Workspace Manager. *windowNum* should have been previously registered with the **registerWindow:toPort:** method.

See also: **– registerWindow:toPort:**

write:

– **write:**(NXTypedStream *)*stream*

Writes the receiving Speaker to the typed stream *stream*.

See also: **– read**

CONSTANTS AND DEFINED TYPES

```
/* File Information */
#define NX_ISFILE           0
#define NX_ISDIRECTORY      1
#define NX_ISAPPLICATION    2
#define NX_ISODMOUNT        3
#define NX_ISNETMOUNT       4
#define NX_ISSCSIMOUNT      5
```

Text

INHERITS FROM	View : Responder : Object
DECLARED IN	appkit/Text.h

CLASS DESCRIPTION

The Text class defines an object that manages text. 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 an NXBrowser. Your application inherits these uses of the Text class when it incorporates any of these objects into its interface. It can also create Text objects for its own purposes.

The Text class is unlike most other classes in the Application Kit in its complexity and range of features. One of its design goals is to provide a comprehensive set of text-handling features so that you'll rarely need to create a subclass. A Text object can (among other things):

- Control the color of its text and background.
- Control the font and layout characteristics of its text.
- Control whether text is editable.
- Wrap text on a word or character basis.
- Write text to, or read it from, an NXStream as either RTF or plain ASCII data.
- Display graphic images within its text.
- Communicate with other applications through the Services menu.
- 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 Text objects.
- Let the user check the spelling of words in its text.
- Let the user control the format of paragraphs by manipulating a ruler.

Interface Builder gives you access to Text objects in several different configurations, such as those found in the TextField, Form, and ScrollView objects in the Palettes window. These classes configure a Text object for a specific purpose. Additionally, all TextFields, Forms, Buttons 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, use a Text object directly.

Plain and Rich Text Objects

When you create a Text object directly, by default it allows only one font, line height, text color, and paragraph format for the entire text. You can set the default font used by new Text instances by sending the Text class object a **setDefaultFont:** message. Once a Text object is created, you can alter its global settings using methods such as

setFont:, **setLineHeight:**, **setTextGray:**, and **setAlignment:**. For convenience, such a Text object will be called a *plain Text object*.

To allow multiple values for these attributes, you must send the Text object a **setMonoFont:**NO message. A Text object that allows multiple fonts also allows multiple paragraph formats, line heights, and so on. Such a Text object can store the content and format of its text by writing RTF (Rich Text Format) data to the pasteboard or to a file. For convenience, such a Text object will be called a *rich Text object*.

In a Text object, each sequence of characters having the same attributes is called a *run*. (See the NXRun structure at the end of this class specification for details.) A Text object in its default state has only one run for the entire text. A rich Text object can have multiple runs. Methods such as **setSelFont:**, **setSelProp:to:**, **setSelGray:**, and **alignSelCenter:** let you programmatically modify the attributes of the selected sequence of characters in a rich Text object. As discussed below, the user can set these attributes by using the Font panel and the ruler.

Text objects are designed to work closely with various objects and services. 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, spelling checker, ruler, and Services menu) take no effort other than deciding whether the service should be enabled or disabled. The following sections discuss these interrelationships.

Notifying the Text Object's Delegate

Many of a Text object's actions can be controlled through an associated object, the Text object's delegate. If it implements any of the following methods, the delegate receives the corresponding message at the appropriate time:

textWillResize:
textDidResize:oldBounds:invalid:
textWillChange:
textDidChange:
textWillEnd:
textDidEnd:endChar:
textDidGetKeys:isEmpty:
textWillSetSel:toFont:
textWillConvert:fromFont:toFont:
textWillWriteRichText:stream:forRun:atPosition:emitDefaultRichText:
textWillReadRichText:stream:atPosition:
textWillStartReadingRichText:
textWillFinishReadingRichText:
textWillWrite:paperSize:
textDidRead:paperSize:

So, for example, if the delegate implements the **textWillChange:** method, it will receive notification upon the user's first attempt to alter the text. Moreover, depending on the method's return value, the delegate can either allow or prohibit changes to the text. (See the section titled "Methods Implemented by the Delegate" for more information.) The delegate can be any object you choose, and one delegate can be used to control multiple Text objects.

Adding Graphics to the Text

A rich Text object allows graphic objects to be embedded in the text. Each object is treated like a single character: The text's line height and character placement are adjusted to accommodate the graphic "character."

In most cases, the graphic object is a subclass of Cell; however, the only requirement is that the embedded object be able to respond to these messages (see the section titled "Methods Implemented by an Embedded Graphic Object" for more information):

```
highlight:inView:lit:
drawSelf:inView:
trackMouse:inRect:ofView:
calcCellSize:
readRichText:forView:
writeRichText:forView:
```

A graphic object can be placed in the text by sending the Text object a **replaceSelWithCell:** message.

A Text object displays a graphic object in its text by sending the object a **drawSelf:inView:** message. To record the object to a file or to the pasteboard, the Text object sends it a **writeRichText:forView:** message. The graphic object must then write an RTF control word along with any data (such as the path of a TIFF file containing its image data) it might need to recreate its image. To reestablish the text containing the graphic image from RTF data, a Text object must know which class to associate with particular RTF control words. You associate a control word with a class object by sending the Text class object a **registerDirective:forClass:** message. Thereafter, whenever a Text object finds the registered control word in RTF data being read from a file or the pasteboard, it will create a new instance of the class and send the object a **readRichText:forView:** message.

Cooperating with Other Objects and Services

Text objects are designed to work with the Application Kit's font conversion system. By default, a Text object keeps the Font panel updated with the font of the current selection. It also changes the font of the selection (for a rich Text object) or of the entire text (for a default Text object) to reflect the user's choices in the Font panel or menu. To disconnect a Text object from this service, send it a **setFontPanelEnabled:**NO message.

If a Text object is a subview of a ScrollView, it can cooperate with the ScrollView to display and update a ruler that displays formatting information. The ScrollView retiles its subviews to make room for the ruler, and the Text object updates the ruler with the format information of the paragraph containing the selection. The **toggleRuler:** method controls the display of this ruler. Users can modify paragraph formats by manipulating the components of the ruler.

By means of the Services menu, a Text object can make use of facilities outside the scope of its own application. By default, a Text object registers with the services system that it can send and receive RTF and plain ASCII data. If the application containing the Text object has a Services menu, a menu item is added for each service provider that can accept or return these formats. To prevent Text objects from registering for services, send the Text class object an **excludeFromServicesMenu:**YES message before any Text objects are created.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Responder	id	nextResponder;
Inherited from View	NXRect	frame;
	NXRect	bounds;
	id	superview;
	id	subviews;
	id	window;
	struct __vFlags	vFlags;

Declared in Text

```
const NXFSM            *breakTable;
const NXFSM            *clickTable;
const unsigned char    *preSelSmartTable;
const unsigned char    *postSelSmartTable;
const unsigned char    *charCategoryTable;
char                   delegateMethods;
NXCharFilterFunc       charFilterFunc;
NXTextFilterFunc       textFilterFunc;
NXTextFunc             scanFunc;
NXTextFunc             drawFunc;
id                     delegate;
int                    tag;
DPSTimedEntry          cursorTE;
NXTextBlock            *firstTextBlock;
NXTextBlock            *lastTextBlock;
NXRunArray             *theRuns;
NXRun                  typingRun;
NXBreakArray           *theBreaks;
int                    growLine;
int                    textLength;
NXCoord                maxY;
NXCoord                maxX;
NXRect                 bodyRect;
NXCoord                borderWidth;
char                   clickCount;
NXSelPt                sp0;
NXSelPt                spN;
NXSelPt                anchorL;
NXSelPt                anchorR;
float                  backgroundGray;
float                  textGray;
float                  selectionGray;
NXSize                 maxSize;
NXSize                 minSize;
struct _tFlags {
    unsigned int           changeState:1;
    unsigned int           charWrap:1;
    unsigned int           haveDown:1;
    unsigned int           anchorIs0:1;
    unsigned int           horizResizable:1;
    unsigned int           vertResizable:1;
    unsigned int           overstrikeDiacriticals:1;
    unsigned int           monoFont:1;
    unsigned int           disableFontPanel:1;
    unsigned int           inClipView:1;
}                      tFlags;
NXStream               *textStream;
```

breakTable	A pointer to the finite-state machine table that specifies word and line breaks.
clickTable	A pointer to the finite-state machine table that defines word boundaries for double-click selection.
preSelSmartTable	A pointer to the table that specifies which characters on the left end of a selection are treated as equivalent to a space.
postSelSmartTable	A pointer to the table that specifies which characters at the right end of a selection are treated as equivalent to a space.
charCategoryTable	A pointer to the table that maps ASCII characters to character classes. Entries are premultiplied by the size of a finite-state machine table entry.
delegateMethods	A record of the notification methods that the delegate implements.
charFilterFunc	The function that checks each character as it's typed into the text.
textFilterFunc	The function that checks the text that's being added to the Text object.
scanFunc	The function that calculates the line of text.
drawFunc	The function that draws the line of text.
delegate	The object that's notified when the Text object is modified.
tag	The integer that the delegate uses to identify the Text object.
cursorTE	The timed-entry number returned by **DPSAddTimedEntry**().
firstTextBlock	A pointer to the first record in a linked list of text blocks.
lastTextBlock	A pointer to the last record in a linked list of text blocks.
theRuns	A pointer to the array of format runs. By default, **theRuns** points to a single run of the default font.
typingRun	The format run to use for the next characters entered.

theBreaks	A pointer to the array of line breaks.
growLine	The line containing the end of the growing selection.
textLength	The number of characters in the Text object.
maxY	The bottom of the last line of text. **maxY** is measured relative to the origin of the **bodyRect**.
maxX	The widest line of text. **maxX** is accurate only after the **calcLine** method is applied.
bodyRect	The rectangle the Text object draws text in.
borderWidth	Reserved for future use.
clickCount	The number of clicks that created the selection.
sp0	The starting position of the selection.
spN	The ending position of the selection.
anchorL	The left anchor position.
anchorR	The right anchor position.
backgroundGray	The background gray value of the text.
textGray	The gray value of the text.
selectionGray	The gray value of the selection.
maxSize	The maximum size of the frame rectangle.
minSize	The minimum size of the frame rectangle.
tFlags.changeState	True if any changes have been made to the text since the Text object became the first responder.
tFlags.charWrap	True if the Text object wraps words whose length exceeds the line length on a character basis. False if such words are truncated at the end of the line.
tFlags.haveDown	True if the left mouse button (or either button if their functions haven't been differentiated) is down.
tFlags.anchorIs0	True if the anchor's position is at **sp0**.
tFlags.horizResizable	True if the Text object's width can grow or shrink.

tFlags.vertResizable	True if the Text object's height can grow or shrink.
tFlags.overstrikeDiacriticals	Reserved for future use.
tFlags.monoFont	True if the Text object uses one font for all its text.
tFlags.disableFontPanel	True if the Text object doesn't update the Font panel automatically.
tFlags.inClipView	True if the Text object is the subview of a ClipView.
textStream	The stream for reading and writing text.

METHOD TYPES

Initializing the class object	+ setDefaultFont: + getDefaultFont + excludeFromServicesMenu: + registerDirective:forClass: + initialize
Initializing a new Text object	– initFrame: – initFrame:text:alignment:
Freeing Text object	– free
Modifying the frame rectangle	– setMaxSize: – getMaxSize: – setMinSize: – getMinSize: – setVertResizable: – isVertResizable – setHorizResizable: – isHorizResizable – sizeTo:: – sizeToFit – resizeText:: – moveTo::

Laying out the text	– setMarginLeft:right:top:bottom:
	– getMarginLeft:right:top:bottom:
	– getMinWidth:minHeight:maxWidth:maxHeight:
	– setAlignment:
	– alignment
	– alignSelLeft:
	– alignSelCenter:
	– alignSelRight:
	– setSelProp:to:
	– changeTabStopAt:to:
	– calcLine
	– setCharWrap:
	– charWrap
	– setNoWrap
	– setParaStyle:
	– defaultParaStyle
	– calcParagraphStyle::
	– setLineHeight:
	– lineHeight
	– setDescentLine:
	– descentLine
Reporting line and position	– lineFromPosition:
	– positionFromLine:
Setting, reading, and writing the text	
	– setText:
	– readText:
	– startReadingRichText
	– readRichText:
	– readRichText:atPosition:
	– finishReadingRichText
	– writeText:
	– writeRichText:
	– writeRichText:from:to:
	– writeRichText:forRun:atPosition: emitDefaultRichText:
	– stream
	– firstTextBlock
	– getParagraph:start:end:rect:
	– getSubstring:start:length:
	– byteLength
	– textLength
Setting editability	– setEditable:
	– isEditable

Editing the text	– copy:
	– copyFont:
	– copyRuler:
	– paste:
	– pasteFont:
	– pasteRuler:
	– cut:
	– delete:
	– clear:
	– selectAll:
	– selectText:
Managing the selection	– subscript:
	– superscript:
	– unscript:
	– underline:
	– showCaret
	– hideCaret
	– setSelectable:
	– isSelectable
	– selectError
	– selectNull
	– setSel::
	– getSel::
	– replaceSel:
	– replaceSel:length:
	– replaceSel:length:runs:
	– replaceSelWithRichText:
	– scrollSelToVisible
Setting the font	– setMonoFont:
	– isMonoFont
	– setFontPanelEnabled:
	– isFontPanelEnabled
	– changeFont:
	– setFont:
	– font
	– setFont:paraStyle:
	– setSelFont:
	– setSelFontFamily:
	– setSelFontSize:
	– setSelFontStyle:
	– setSelFont:paraStyle:
Checking spelling	– checkSpelling:
	– showGuessPanel:
Managing the ruler	– toggleRuler:
	– isRulerVisible

Modifying graphic attributes	– setBackgroundGray:
	– backgroundGray
	– setBackgroundColor:
	– backgroundColor
	– setSelGray:
	– selGray
	– setSelColor:
	– setTextGray:
	– textGray
	– setTextColor:
	– textColor
Reusing a Text object	– renewFont:text:frame:tag:
	– renewFont:size:style:text:frame:tag:
	– renewRuns:text:frame:tag:
	– windowChanged:
Displaying	– drawSelf::
	– setRetainedWhileDrawing:
	– isRetainedWhileDrawing
Assigning a tag	– setTag:
	– tag
Handling event messages	– acceptsFirstResponder
	– becomeFirstResponder
	– resignFirstResponder
	– becomeKeyWindow
	– resignKeyWindow
	– mouseDown:
	– keyDown:
	– moveCaret:
Displaying graphics within the text	
	+ registerDirective:forClass:
	– replaceSelWithCell:
	– replaceSelWithView:
	– setLocation:ofCell:
	– getLocation:ofCell:
	– getLocation:ofView:
Using the Services menu	+ excludeFromServicesMenu:
	– validRequestorForSendType: andReturnType:
	– readSelectionFromPasteboard:
	– writeSelectionToPasteboard:types:

Setting tables and functions	– setCharFilter: – charFilter – setTextFilter: – textFilter – setBreakTable: – breakTable – setPreSelSmartTable: – preSelSmartTable – setPostSelSmartTable: – postSelSmartTable – setCharCategoryTable: – charCategoryTable – setClickTable: – clickTable – setScanFunc: – scanFunc – setDrawFunc: – drawFunc
Printing	– adjustPageHeightNew:top:bottom:limit:
Archiving	– read: – write:
Assigning a delegate	– setDelegate: – delegate

CLASS METHODS

excludeFromServicesMenu:

+ **excludeFromServicesMenu:**(BOOL)*flag*

Controls whether Text objects will communicate with interapplication services through the Services menu. By default, as each new Text instance is initialized, it registers with the Application object that it's capable of sending and receiving the pasteboard types identified by NXAsciiPboardType and NXRTFPboardType. If you want to prevent Text objects in your application from registering for services that can receive and send these types, send the Text class object an **excludeFromServicesMenu:**YES message. If, for example, your application displays text but doesn't have editable text fields, you might use this method.

Send an **excludeFromServicesMenu:** message early in the execution of your application, either before sending the Application object a **run** message or in the Application delegate's **appWillInit:** method. Returns **self**.

See also: – **validRequestorForSendType:andReturnType:**,
– **registerServicesMenuSendTypes:andReturnTypes:** (Application)

getDefaultFont

+ getDefaultFont

Returns the Font object that corresponds to the Text object's default. Unless you've changed the default font by sending a **setDefaultFont:** message, or taken advantage of the NXFont parameter using defaults, **getDefaultFont** returns a Font object for a 12-point Helvetica font with a flipped font matrix.

See also: + **setDefaultFont:**, – **setFont:**

initialize

+ initialize

Initializes the class object. The **initialize** message is sent for you before the class object receives any other message; you never send an **initialize** message directly. Returns **self**.

See also: + **initialize** (Object)

registerDirective:forClass:

+ **registerDirective:**(const char *)*directive* **forClass:***class*

Creates an association in the Text class object between the RTF control word *directive* and *class*, a class object. Thereafter, when a Text instance encounters *directive* while reading a stream of RTF text, it creates a new *class* instance. The new instance is sent a **readRichText:forView:** message to let it read its image data from the RTF text. Conversely, when a Text object is writing RTF data to a stream and encounters an object of the *class* class, the Text object sends the object a **writeRichText:forView:** message to let it record its representation in the RTF text. Thus, this method is instrumental in enabling a Text object to read, display, and write an image within a text stream.

An object of the *class* class must implement these methods:

highlight:inView:lit:
drawSelf:inView:
trackMouse:inRect:ofView:
calcCellSize:
readRichText:forView:
writeRichText:forView:

See the section titled "Methods Implemented by an Embedded Graphic Object" for more information on these methods.

Returns **nil** if *directive* or *class* has already been registered; otherwise, returns **self**.

See also: – **replaceSelWithCell:**

setDefaultFont:

+ **setDefaultFont:***anObject*

Sets the default font for the Text class object. The argument passed to this method is the **id** of the Font object for the desired font. Since a Text object uses a flipped coordinate system, make sure the Font object you specify uses a matrix that flips the y-axis of the characters. Returns *anObject*.

See also: + **getDefaultFont**, – **setLineHeight:**, + **newFont:size:** (Font)

INSTANCE METHODS

acceptsFirstResponder

– (BOOL)**acceptsFirstResponder**

Assuming the text is selectable, returns YES to let the Text object become the first responder; otherwise, returns NO. **acceptsFirstResponder** messages are sent for you; you never send them yourself.

See also: – **setSelectable:**, – **setDelegate:**, – **resignFirstResponder**

adjustPageHeightNew:top:bottom:limit:

– **adjustPageHeightNew:**(float *)*newBottom*
  top:(float)*oldTop*
  bottom:(float)*oldBottom*
  limit:(float)*bottomLimit*

During automatic pagination, this method is performed to help lay a grid of pages over the top-level view being printed. *newBottom* is passed in undefined and must be set by this method. *oldTop* and *oldBottom* are the current values for the horizontal strip being created. *bottomLimit* is the topmost value *newBottom* can be set to. If this limit is broken, the new value is ignored. By default, this method tries to prevent the view from being cut in two. All parameters are in the view's own coordinate system. Returns **self**.

alignment

– (int)**alignment**

Returns a value indicating the default alignment of the text. The returned value is equal to one of these constants:

Constant	**Alignment**
NX_LEFTALIGNED	Flush to left edge of the **bodyRect**.
NX_RIGHTALIGNED	Flush to right edge of the **bodyRect**.
NX_CENTERED	Each line centered between left and right edges of the **bodyRect**.
NX_JUSTIFIED	Flush to left and right edges of the **bodyRect**; justified. Not yet implemented.

See also: – **setAlignment:**

alignSelCenter:

– **alignSelCenter:***sender*

Sets the paragraph style of one or more paragraphs so that text is centered between the left and right margins. For a plain Text object, all paragraphs are affected. For a rich Text object, only those paragraphs marked by the selection are affected. The sending object passes its **id** as part of the **alignSelCenter:** message. The text is rewrapped and redrawn. Returns **self**.

See also: – **alignSelLeft:**, – **alignSelRight:**, – **setSelProp:to:**, – **setMonoFont:**

alignSelLeft:

– **alignSelLeft:***sender*

Sets the paragraph style of one or more paragraphs so that text is aligned to the left margin. For a plain Text object, all paragraphs are affected. For a rich Text object, only those paragraphs marked by the selection are affected. The sending object passes its **id** as part of the **alignSelLeft:** message. The text is rewrapped and redrawn. Returns **self**.

See also: – **alignSelCenter:**, – **alignSelRight:**, – **setSelProp:to:**, – **setMonoFont:**

alignSelRight:

– **alignSelRight:***sender*

Sets the paragraph style of one or more paragraphs so that text is aligned to the right margin. For a plain Text object, all paragraphs are affected. For a rich Text object, only those paragraphs marked by the selection are affected. The sending object passes its **id** as part of the **alignSelRight:** message. The text is rewrapped and redrawn. Returns **self**.

See also: – **alignSelCenter:**, – **alignSelLeft:**, – **setSelProp:to:**, – **setMonoFont:**

backgroundColor

– (NXColor)**backgroundColor**

Returns the background color of the text.

See also: – **setBackgroundGray:**, – **backgroundGray:**, – **setBackgroundColor:**, – **setTextGray:**, – **textGray**, – **setTextColor:**, – **textColor**, – **setSelGray:**, – **selGray**, – **setSelColor:**

backgroundGray

– (float)**backgroundGray**

Returns the gray value of the text's background.

See also: – **setBackgroundGray:**, – **setBackgroundColor:**, – **backgroundColor**, –**setTextGray:**, – **textGray**, – **setTextColor:**, – **textColor**, – **setSelGray:**, – **selGray**, – **setSelColor:**

becomeFirstResponder

– **becomeFirstResponder**

Lets the Text object know that it's becoming the first responder. By default, the Text object always accepts becoming first responder. **becomeFirstResponder** messages are sent for you; you never send them yourself. Returns **self**.

See also: – **setDelegate:**, –**acceptsFirstResponder**, – **selectError**

becomeKeyWindow

– **becomeKeyWindow**

Activates the caret if it exists. **becomeKeyWindow** messages are sent by an application's Window object, which, upon receiving a mouse-down event, sends a **becomeKeyWindow** message to the first responder. You should never directly send this message to a Text object. Returns **self**.

See also: – **showCaret**, – **hideCaret**, – **becomeKeyWindow** (Window)

breakTable

– (const NXFSM *)**breakTable**

Returns a pointer to the break table, the finite-state machine table that the Text object uses to determine word boundaries.

See also: – **setBreakTable:**

byteLength

– (int)**byteLength**

Returns the number of bytes used by the characters in the receiving Text object. The number doesn't include the null terminator ('\0') that **getSubstring:start:length:** returns if you ask for all the text in a Text object.

In a standard Text object, the number of bytes is equal to the number of characters. Subclasses of Text that use more than one byte per character should override this method to return an accurate count of the number of bytes used to store the text.

See also: – **textLength**, – **getSubstring:start:length:**

calcLine

– (int)**calcLine**

Calculates the array of line breaks for the text. The text will then be redrawn if autodisplay is set.

This message should be sent after the Text object's frame is changed. These methods send a **calcLine** message as part of their implementation:

– initFrame:text:alignment:	– readText:
– read:	– renewFont:size:style:text:frame:tag:
– renewFont:text:frame:tag:	– setFont:
– renewRuns:text:frame:tag:	– setParaStyle:
– setFont:paraStyle:	– setText:

In addition, if a vertically resizable Text object is the document view of a ScrollView, and the ScrollView is resized, the Text object receives a **calcLine** message. Has no significant return value.

See also: – **readText:**, – **renewRuns:text:frame:tag:**

calcParagraphStyle::

– (void *)**calcParagraphStyle:***fontId* **:**(int)*alignment*

Recalculates the default paragraph style given the Font's *fontId* and *alignment*. The Text object sends this message for you after its font has been changed; you will rarely need to send a **calcParagraphStyle::** message directly. Returns a pointer to an NXTextStyle structure that describes the default style.

See also: – **defaultParaStyle**

changeFont:

– **changeFont:***sender*

Changes the font of the selection for a rich Text object. It changes the font for the entire Text object for a plain Text object. *sender* must respond to the **convertFont:** message.

If the Text object's delegate implements the method, it receives a **textWillConvert:fromFont:toFont:** notification message for each text run that's about to be converted.

See also: – **setFontPanelEnabled:**

changeTabStopAt:to:

– **changeTabStopAt:**(NXCoord)*oldX* **to:**(NXCoord)*newX*

Moves the tab stop from the receiving Text object's x coordinate *oldX* to the coordinate *newX*. For a plain Text object, all paragraphs are affected. For a rich Text object, only those paragraphs marked by the selection are affected. The text is rewrapped and redrawn. Returns **self**.

See also: – **setMonoFont:**, – **setSelProp:to:**

charCategoryTable

– (const unsigned char *)**charCategoryTable**

Returns a pointer to the character category table, the table that maps ASCII characters to character categories.

See also: – **setCharCategoryTable:**

charFilter

– (NXCharFilterFunc)**charFilter**

Returns the character filter function, the function that analyzes each character the user enters. By default, this function is **NXEditorFilter()**.

See also: – **setCharFilter:**

charWrap

– (BOOL)**charWrap**

Returns **charWrap**, a flag indicating how words whose length exceeds the line length should be treated. If YES, long words are wrapped on a character basis. If NO, long words are truncated at the boundary of the **bodyRect**.

See also: – **setCharWrap:**

checkSpelling:

– **checkSpelling:***sender*

Searches for a misspelled word in the text of the receiving Text object. The search starts at the current 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 or listed in the user's local dictionary in **~/.NeXT/LocalDictionary**, it's highlighted. A **showGuessPanel:** message will then display the Guess panel and allow the user to make a correction or add the word to the local dictionary. Returns **self**.

See also: – **showGuessPanel:**

clear:

– **clear:***sender*

Provided for backward compatibility. Use the **delete:** method instead.

See also: – **delete:**

clickTable

– (const NXFSM *)**clickTable**

Returns a pointer to the click table, the finite-state machine table that defines word boundaries for double-click selection.

See also: – **setClickTable:**

copy:

– **copy:***sender*

Copies the selected text from the Text object to the selection pasteboard. The selection remains unchanged. The pasteboard receives the text and its corresponding run information. The pasteboard types used are NXAsciiPboardType and NXRTFPboardType.

The sender passes its **id** as part of the **copy:** message. Returns **self**.

See also: **– cut:**, **– paste:**, **– delete:**, **– copyFont:**, **– pasteFont:**, **– copyRuler:**, **– pasteRuler:**

copyFont:

– **copyFont:***sender*

Copies font information for the selected text to the font pasteboard. If the selection spans more than one font, the information copied is that of the first font in the selection. The selection remains unchanged. The pasteboard type used is NXFontPboardType.

The sender passes its **id** as the argument of the **copyFont:** message. Returns **self**.

See also: **– pasteFont:**, **– copyRuler:**, **– pasteRuler:**, **– copy:**, **– cut:**, **– paste:**, **– delete:**

copyRuler:

– **copyRuler:***sender*

Copies ruler information for the paragraph containing the selection to the ruler pasteboard. The selection expands to paragraph boundaries.

The ruler controls a paragraph's text alignment, tab settings, and indentation. If the selection spans more than one paragraph, the information copied is that of the first paragraph in the selection. The pasteboard type used is NXRulerPboardType.

Once copied to the pasteboard, ruler information can be pasted into another object or application that's able to paste RTF data into its document.

The sender passes its **id** as the argument of the **copyRuler:** message. Returns **self**.

See also: **– pasteRuler:**, **– copyFont:**, **– pasteFont:**, **– copy:**, **– cut:**, **– paste:**, **– delete:**

cut:

– **cut:***sender*

Copies the selected text to the pasteboard and then deletes it from the Text object. The pasteboard receives the text and its corresponding font information.

If the Text object's delegate implements the method, it receives a **textDidGetKeys:isEmpty:** message immediately after the cut operation. If this is the first change since the Text object became the first responder (and the delegate implements the method), a **textDidChange:** message is also sent to the delegate.

The *sender* passes its **id** as part of the **cut:** message. Returns **self**.

See also: – **copy:**, – **paste:**, – **delete:**, – **textDidGetKeys:isEmpty:**, – **textDidChange:**

defaultParaStyle

– (void *)**defaultParaStyle**

Returns by reference the default paragraph style for the text. The pointer that's returned refers to an NXTextStyle structure. The fields of this structure contain default paragraph indentation, alignment, line height, descent line, and tab information. The Text object's default values for these attributes can be altered using methods such as **setParaStyle:**, **setAlignment:**, **setLineHeight:**, and **setDescentLine:**.

See also: – **setParaStyle:**, – **setAlignment:**, – **setLineHeight:**, – **setDescentLine:**

delegate

– **delegate**

Returns the Text object's delegate.

See also: – **setDelegate:**

delete:

– **delete:***sender*

Deletes the selection without adding it to the pasteboard. The sender passes its **id** as part of the **delete:** message.

If the Text object's delegate implements the method, it receives a **textDidGetKeys:isEmpty:** message immediately after the delete operation. If this is the first change since the Text object became the first responder (and the delegate implements the method), a **textDidChange:** message is also sent to the delegate.

The **delete:** method replaces **clear:**. Returns **self**.

See also: – **cut:**, – **copy:**, – **paste:**, – **textDidGetKeys:isEmpty:**, – **textDidChange:**

descentLine

– (NXCoord)**descentLine**

Returns the default descent line for the Text object. The descent line is the distance from the bottom of a line of text to the base line of the text.

See also: – **setDescentLine:**

drawFunc

– (NXTextFunc)**drawFunc**

Returns the draw function, the function that's called to draw each line of text. **NXDrawALine()** is the default draw function.

See also: – **setDrawFunc:**, – **setScanFunc:**

drawSelf::

– **drawSelf:**(const NXRect *)*rects* **:**(int)*rectCount*

Draws a Text object. You never send a **drawSelf::** message directly, although you may want to override this method to change the way a Text object draws itself. Returns **self**.

See also: – **drawSelf::** (View)

finishReadingRichText

– **finishReadingRichText**

Notifies the Text object that it has finished reading RTF data. The Text object responds by sending its delegate a **textWillFinishReadingRichText:** message, assuming there is a delegate and it responds to this message. The delegate can then perform any required cleanup. Alternatively, a subclass of Text could put these cleanup routines in its own implementation of this method. Returns **self**.

firstTextBlock

– (NXTextBlock *)**firstTextBlock**

Returns a pointer to the first text block. You can traverse this head of the linked list of text blocks to read the contents of the Text object. In most cases, however, it's better to use the **getSubstring:start:length:** method to get a substring of the text or the **stream** method to get read-only access to the entire contents of the Text object.

See also: – **getSubstring:start:length:**, – **stream**

font

– **font**

Returns the Font object for a plain Text object. For rich Text objects, the Font object for the first text run is returned.

See also: – **setFont:**

free

– **free**

Releases the storage for a Text object.

See also: – **free** (View)

getLocation:ofCell:

– **getLocation:**(NXPoint *)*origin* **ofCell:***cell*

Places the x and y coordinates of *cell* in the NXPoint structure specified by *origin*. The coordinates are in the Text object's coordinate system. *cell* is a Cell object that's displayed as part of the text.

Returns **nil** if the Cell object isn't part of the text; otherwise, returns **self**.

See also: – **replaceSelWithCell:**, – **setLocation:ofCell:**, – **getLocation:ofView:**, – **calcCellSize:** (Cell)

getLocation:ofView:

– **getLocation:**(NXPoint *)*origin* **ofView:***view*

Unimplemented.

getMarginLeft:right:top:bottom:

– **getMarginLeft:**(NXCoord *)*leftMargin*
 right:(NXCoord *)*rightMargin*
 top:(NXCoord *)*topMargin*
 bottom:(NXCoord *)*bottomMargin*

Calculates the dimensions of the Text object's margins and returns by reference these values in its four arguments. Returns **self**.

See also: – **setMarginLeft:right:top:bottom:**

getMaxSize:

– **getMaxSize:**(NXSize *)*theSize*

Copies the maximum size of the Text object into the structure referred to by *theSize*. Returns **self**.

See also: – **setMaxSize:**, – **getMinSize:**

getMinSize:

– **getMinSize:**(NXSize *)*theSize*

Copies the minimum size of the Text object into the structure referred to by *theSize*. Returns **self**.

See also: – **setMinSize:**, – **getMaxSize:**

getMinWidth:minHeight:maxWidth:maxHeight:

– **getMinWidth:**(NXCoord *)*width*
 minHeight:(NXCoord *)*height*
 maxWidth:(NXCoord)*widthMax*
 maxHeight:(NXCoord)*heightMax*

Calculates the minimum width and height needed to contain the text. Given a maximum width and height (*widthMax* and *heightMax*), this method copies the minimum width and height to the addresses pointed to by the *width* and *height* arguments. This method doesn't rewrap the text. To get the absolute minimum dimensions of the text, send a **getMinWidth:minHeight:maxWidth:maxHeight:** message only after sending a **calcLine** message.

The values derived by this method are accurate only if the Text object hasn't been scaled. Returns **self**.

See also: – **sizeToFit**

getParagraph:start:end:rect:

– **getParagraph:**(int)*prNumber*
  start:(int *)*startPos*
  end:(int *)*endPos*
  rect:(NXRect *)*paragraphRect*

Copies the positions of the first and last characters of the specified paragraph to the addresses *startPos* and *endPos*. It also copies the paragraph's bounding rectangle into the structure referred to by *paragraphRect*. A paragraph ends in a Return character; the first paragraph is paragraph 0, the second is paragraph 1, and so on. Returns **self**.

See also: – **getSubstring:start:length:**, – **firstTextBlock**

getSel::

– **getSel:**(NXSelPt *)*start* **:**(NXSelPt *)*end*

Copies the starting and ending character positions of the selection into the addresses referred to by *start* and *end*. *start* points to the beginning of the selection; *end* points to the end of the selection. Returns **self**.

See also: – **setSel::**

getSubstring:start:length:

– (int)**getSubstring:**(char *)*buf*
  start:(int)*startPos*
  length:(int)*numChars*

Copies a substring of the text to a specified memory location. The substring is specified by *startPos* and *numChars*. *startPos* is the position of the first character of the substring; *numChars* is the number of characters to be copied. *buf* is the starting address of the memory location for the substring. **getSubstring:start:length:** returns the number of characters actually copied. This number may be less than *numChars* if the last character position is less than *startPos* + *numChars*. Returns -1 if *startPos* is beyond the end of the text.

getSubstring:start:length: appends a null terminator ('\0') to the substring only if the requested substring includes the end of the Text object's text.

See also: – **textLength**, – **getSel::**

hideCaret

– **hideCaret**

Removes the caret from the text. The Text object sends itself **hideCaret** messages whenever the display of the caret would be inappropriate; you rarely need to send a **hideCaret** message directly. Occasions when the **hideCaret** message is sent include whenever the Text object receives a **resignKeyWindow**, **mouseDown:**, or **keyDown:** message. Returns **self**.

See also: – **showCaret**

initFrame:

– **initFrame:**(const NXRect *)*frameRect*

Initializes a new Text object. This method invokes the **initFrame:text:alignment:** method with the size and location specified by *frameRect*. Text alignment is set to NX_LEFTALIGNED. Returns **self**.

See also: – **initFrame:text:alignment:**

initFrame:text:alignment:

– **initFrame:**(const NXRect *)*frameRect*
    text:(const char *)*theText*
    alignment:(int)*mode*

Initializes a new Text object. This is the designated initializer for Text objects: If you subclass Text, your subclass's designated initializer must maintain the initializer chain by sending a message to **super** to invoke this method. See the introduction to the class specifications for more information.

The three arguments specify the Text object's frame rectangle, its text, and the alignment of the text. The *frameRect* argument specifies the Text object's location and size in its superview's coordinates. A Text object's superview must be a flipped view that's neither scaled nor rotated. The second argument, *theText*, is a null-terminated array of characters. The *mode* argument determines how the text is drawn with respect to the **bodyRect**:

Constant	**Alignment**
NX_LEFTALIGNED	Flush to left edge of the **bodyRect**.
NX_RIGHTALIGNED	Flush to right edge of the **bodyRect**.
NX_CENTERED	Each line centered between left and right edges of the **bodyRect**.
NX_JUSTIFIED	Flush to left and right edges of the **bodyRect**; justified. Not yet implemented.

The Text object returned by this method uses the class object's default font (see **setDefaultFont:**) and uses **NXEditorFilter()** as its character filter. It wraps words whose length exceeds the line length. It sets its View properties to draw in its superview, to be flipped, and to be transparent. For more efficient editing, you can send a **setOpaque:** message to make the Text object opaque.

Text editing is designed to work in buffered windows only. In a nonretained or retained window, editing text in a Text object causes flickering. (However, to get better drawing performance without causing flickering during editing, see **setRetainedWhileDrawing:**).

Returns **self**.

See also: – **initFrame:**

isEditable

– (BOOL)**isEditable**

Returns YES if the text can be edited, NO if not.

See also: – **isSelectable**, – **setDelegate:**

isFontPanelEnabled

– (BOOL)**isFontPanelEnabled**

Returns YES if the Text object will respond to the Font panel, NO if not. The default value is YES.

See also: – **setFontPanelEnabled:**

isHorizResizable

– (BOOL)**isHorizResizable**

Returns YES if the text can automatically change size horizontally, NO if not. The default value is NO.

See also: – **setVertResizable:**, – **isVertResizable**, – **setHorizResizable:**

isMonoFont

– (BOOL)**isMonoFont**

Returns YES if the Text object allows multiple paragraph styles and fonts, NO if not.

See also: – **setMonoFont:**

isRetainedWhileDrawing

– (BOOL)**isRetainedWhileDrawing**

Returns YES if the Text object automatically changes its window's buffering type from buffered to retained whenever it redraws itself, NO if not.

See also: **– setRetainedWhileDrawing:**, **– drawSelf::**

isRulerVisible

– (BOOL)**isRulerVisible**

Returns YES if the ruler is visible in the Text object's superview, a ScrollView; otherwise, returns NO.

See also: **– toggleRuler:**

isSelectable

– (BOOL)**isSelectable**

Returns YES if the text can be selected, NO if not.

See also: **– isEditable**, **– setDelegate:**

isVertResizable

– (BOOL)**isVertResizable**

Returns YES if the text can automatically change size vertically, NO if not. The default value is NO.

See also: **– setVertResizable:**, **– setHorizResizable:**, **– isHorizResizable**

keyDown:

– **keyDown:**(NXEvent *)*theEvent*

Analyzes key-down events received by the Text object. **keyDown:** first uses the Text object's character filter function to determine whether the event should be interpreted as a command to move the cursor or as a command to end the Text object's status as the first responder. If the latter, the Text object's delegate is given an opportunity to prevent the change.

If the event represents a character that should be added to the text, the Text object sets up a modal event loop to process it along with other key-down events as they're received. The text is redrawn, and then **keyDown:** notifies the delegate that the text has changed. This message is sent by the system in response to keyboard events. You never send this message, though you may want to override it.

See also: **– setCharFilter:**, **– setDelegate:**, **– getNextEvent:waitFor:** (Application)

lineFromPosition:

– (int)**lineFromPosition:**(int)*position*

Returns the line number that contains the character at *position*. To get more information about the contents of the Text object, use the stream returned by the **stream** method to read the contents of the Text object.

See also: – **positionFromLine:**, – **stream**

lineHeight

– (NXCoord)**lineHeight**

Returns the default line height for the Text object.

See also: – **setLineHeight:**

mouseDown:

– **mouseDown:**(NXEvent *)*theEvent*

Responds to mouse-down events. When a Text object that allows selection receives a **mouseDown:** message, it tracks mouse-dragged events and responds by adjusting the selection and autoscrolling, if necessary. You never send this message, though you may want to override it.

See also: – **setEditable:**, – **setDelegate:**, – **getNextEvent:waitFor:** (Application)

moveCaret:

– **moveCaret:**(unsigned short)*theKey*

Moves the caret either left, right, up, or down if *theKey* is NX_LEFT, NX_RIGHT, NX_UP, or NX_DOWN. If *theKey* isn't one of these four values, the caret doesn't move. Returns **self**.

See also: – **keyDown:**

moveTo::

– **moveTo:**(NXCoord)*x* **:**(NXCoord)*y*

Moves the origin of the Text object's frame rectangle to (*x*, *y*) in its superview's coordinates. Returns **self**.

See also: – **moveTo::** (View)

overstrikeDiacriticals

– (int)**overstrikeDiacriticals**

Unimplemented.

paste:

– **paste:***sender*

Places the contents of the selection pasteboard into the Text object at the position of the current selection. If the selection is zero-width, the text is inserted at the caret. If the selection has positive width, the selection is replaced by the contents of the pasteboard. In either case, the text is rewrapped and redrawn.

Before the paste operation, a **textDidChange:** message is sent to the delegate, assuming that this is the first change since the Text object became the first responder and that the delegate implements the method. After the paste operation, the delegate receives a **textDidGetKeys:isEmpty:** message, if it implements the method.

sender is the **id** of the sending object. **paste:** returns **nil** if the pasteboard can provide neither NXAsciiPboardType nor NXRTFPboardType format types; otherwise, returns **self**.

See also: – **copy:**, – **cut:**, – **delete:**, – **copyFont:**, – **copyRuler:**, – **pasteFont:**, – **pasteRuler:**, – **textDidGetKeys:isEmpty:**, – **textDidChange:**

pasteFont:

– **pasteFont:***sender*

Takes font information from the font pasteboard and applies it to the current selection. If the selection is zero-width, only those characters subsequently entered at the insertion point are affected.

pasteFont: works only with rich Text objects (see **setMonoFont:**). Attempting to paste a font into a plain Text object generates a system beep without altering any fonts.

Before the paste operation, a **textDidChange:** message is sent to the delegate, assuming that this is the first change since the Text object became the first responder and that the delegate implements the method. After the paste operation, the delegate receives a **textDidGetKeys:isEmpty:** message, if it implements the method.

sender is the **id** of the sending object. After the font is pasted, the text is rewrapped and redrawn. **pasteFont:** returns **nil** if the pasteboard has no data of the type NXFontPboardType; otherwise, returns **self**.

See also: – **copyFont:**, – **copyRuler:**, – **pasteRuler:**, – **copy:**, – **cut:**, – **delete:**, – **paste:**, – **setMonoFont:** – **textDidGetKeys:isEmpty:**, – **textDidChange:**

pasteRuler:

– **pasteRuler:***sender*

Takes ruler information from the ruler pasteboard and applies it to the paragraph or paragraphs marked by the current selection. The ruler controls a paragraph's text alignment, tab settings, and indentation.

pasteRuler: works only with rich Text objects (see **setMonoFont:**). Attempting to paste a ruler into a plain Text object generates a system beep without altering any ruler settings.

Before the paste operation, a **textDidChange:** message is sent to the delegate, assuming that this is the first change since the Text object became the first responder and that the delegate implements the method. After the paste operation, the delegate receives a **textDidGetKeys:isEmpty:** message, if it implements the method.

sender is the **id** of the sending object. After the ruler is pasted, the text is rewrapped and redrawn. If the ruler is visible, it's also updated. **pasteRuler:** returns **nil** if the pasteboard has no data of the type NXRulerPboardType; otherwise, returns **self**.

See also: – **copyRuler:**, – **copyFont:**, – **pasteFont:**, – **copy:**, – **cut:**, – **delete:**, – **paste:**, – **setMonoFont:** – **textDidGetKeys:isEmpty:**, – **textDidChange:**

positionFromLine:

– (int)**positionFromLine:**(int)*line*

Returns the character position of the line numbered *line*. Each line is terminated by a Return character, and the first line in a Text object is line 1. To find the length of a line, you can send the **positionFromLine:** message with two successive lines, and use the difference of the two to get the line length. To get more information about the contents of the Text object, use the stream returned by the **stream** method to read the contents of the Text object.

See also: – **lineFromPosition:**, – **stream**

postSelSmartTable

– (const unsigned char *)**postSelSmartTable**

Returns a pointer to the table that specifies which characters on the right end of a selection are treated as equivalent to a space character.

See also: – **setPostSelSmartTable:**, – **setPreSelSmartTable:**, – **preSelSmartTable**

preSelSmartTable

– (const unsigned char *)**preSelSmartTable**

Returns a pointer to the table that specifies which characters on the left end of a selection are treated as equivalent to a space character.

See also: – **setPreSelSmartTable:**, – **setPostSelSmartTable:**, – **postSelSmartTable**

read:

– **read:**(NXTypedStream *)*stream*

Reads the Text object in from the typed stream *stream*. A **read:** message is sent in response to archiving; you never send this message directly. Returns **self**.

readRichText:

– **readRichText:**(NXStream *)*stream*

Reads RTF text from *stream* into the Text object and formats the text accordingly. The Text object is resized to be large enough for all the text to be visible. The *NeXTstep Concepts* manual lists the RTF directives that the Text object understands. RTF directives that aren't implemented are ignored. Returns **self**.

See also: – **writeRichText:**

readRichText:atPosition:

– **readRichText:**(NXStream *)*stream* **atPosition:**(int)*position*

Reads RTF text from *stream* into the Text object's text at *position* and formats the text accordingly. You never send this message, but may want to override it to read special RTF directives while the Text object is reading RTF data. If there is a delegate, and it implements the method, the Text object sends it a **textWillReadRichText:atPosition** message. Returns **self**.

readSelectionFromPasteboard:

– **readSelectionFromPasteboard:***pboard*

Replaces the current selection with data from the supplied Pasteboard object, *pboard*. When the user chooses a command in the Services menu, a **writeSelectionToPasteboard:types:** message is sent to the first responder. This message is followed by a **readSelectionFromPasteboard:** message, if the command requires the requesting application to replace its selection with data from the service provider.

See also: – **writeSelectionToPasteboard:types:**,
– **validRequestorForSendType:andReturnTypes:**

readText:

– **readText:**(NXStream *)*stream*

Reads new text into the Text object from *stream.* All previous text is deleted. The Text object wraps and redraws the new text if autodisplay is enabled. This method doesn't affect the object's frame or bounds rectangle. To resize the text rectangle to make the text entirely visible, use the **sizeToFit** method. Returns **self**. This method raises an NX_textBadRead exception if an error occurs while reading from *stream.*

See also: – **setSel::**, – **setText:**, – **readRichText:**, – **sizeToFit**

renewFont:size:style:text:frame:tag:

– **renewFont:**(const char *)*newFontName*
    size:(float)*newFontSize*
    style:(int)*newFontStyle*
    text:(const char *)*newText*
    frame:(const NXRect *)*newFrame*
    tag:(int)*newTag*

Resets a Text object so that it can be reused to draw or edit another piece of text. If *newText* is NULL, the new text is the same as the previous text. *newTag* sets the Text object's tag. A font object is created with *newFontName*, *newFontSize*, and *newFontStyle*. This method is a convenient cover for the **renewRuns:text:frame:tag:** method. Returns **self**.

See also: – **renewRuns:text:frame:tag:**, – **setText:**

renewFont:text:frame:tag:

– **renewFont:***newFontId*
    text:(const char *)*newText*
    frame:(const NXRect *)*newFrame*
    tag:(int)*newTag*

Resets a Text object so that it can be reused to draw or edit another piece of text. If *newText* is NULL, the new text is the same as the previous text. *newTag* sets a Text object's tag. This method is a convenient cover for the **renewRuns:text:frame:tag:** method. Returns **self**.

See also: – **setText:**

renewRuns:text:frame:tag:

– **renewRuns:**(NXRunArray *)*newRuns*
  text:(const char *)*newText*
  frame:(const NXRect *)*newFrame*
  tag:(int)*newTag*

Resets a Text object so that it can be reused to draw or edit another piece of text. If *newRuns* is NULL, the new text uses the same runs as the previous text. If *newText* is NULL, the new text is the same as the previous text. *newTag* sets a Text object's tag. Returns **self**.

See also: – **setText:**

replaceSel:

– **replaceSel:**(const char *)*aString*

Replaces the current selection with text from *aString*, a null-terminated character string, and then rewraps and redisplays the text. Returns **self**.

See also: – **replaceSel:length:**

replaceSel:length:

– **replaceSel:**(const char *)*aString* **length:**(int)*length*

Replaces the current selection with *length* characters of text from *aString*, and then rewraps and redisplays the text. Returns **self**.

See also: – **replaceSel:**

replaceSel:length:runs:

– **replaceSel:**(const char *)*aString*
  length:(int)*length*
  runs:(NXRunArray *)*insertRuns*

Replaces the current selection with *length* characters of text from *aString*, using *insertRuns* to describe the run changes. Another way to replace the selection with multiple-run text is with **replaceSelWithRichText:**.

After replacing the selection, this method rewraps and redisplays the text. Returns **self**.

See also: – **replaceSel:**, – **replaceSelWithRichText:**

replaceSelWithCell:

– **replaceSelWithCell:***cell*

Replaces the current selection with the image provided by *cell.* This method works only with rich Text objects. (See **setMonoFont:**.)

The image is treated like a single character. Its height and width are determined by sending the Cell a **calcCellSize:** message. The height determines the line height of the line containing the image, and the width sets the character placement in the line. The image is drawn by sending the Cell a **drawSelf:inView:** message.

After receiving a **replaceSelWithCell:** message, a Text object rewraps and redisplays its contents. Returns **self**.

See also: – **setMonoFont:**, – **calcCellSize:** (Cell), – **drawSelf:inView:** (Cell)

replaceSelWithRichText:

– **replaceSelWithRichText:**(NXStream *)*stream*

Replaces the current selection with RTF data from *stream.* A **replaceSelWithRichText:** message is sent in response to pasting RTF data from the pasteboard.

After replacing the selection, this method rewraps and redisplays the text. Returns **self.**

See also: – **replaceSel:**, – **replaceSel:length:runs:**

replaceSelWithView:

– **replaceSelWithView:***view*

Unimplemented.

resignFirstResponder

– **resignFirstResponder**

Asks the Text object's delegate for permission before letting the Text object cease being the first responder. If the delegate's **textWillEnd:** method returns a nonzero value, the Text object remains the first responder, the entire text becomes the selection, and this method returns **nil**. Otherwise, **resignFirstResponder** returns **self.**

resignFirstResponder messages are sent for you; you never send them yourself.

See also: – **setDelegate:**, –**acceptsFirstResponder**, – **selectError**

resignKeyWindow

– **resignKeyWindow**

Deactivates the caret when the Text object's window ceases to be the key window. A Window, before it ceases to be the application's key window, sends this message to its first responder. You should never directly send this message to a Text object. Returns **self.**

See also: – **becomeKeyWindow**

resizeText::

– **resizeText:**(const NXRect *)*oldBounds* **:**(const NXRect *)*maxRect*

Causes the superview to redraw exposed portions of itself after the Text object's frame has changed in response to editing. You never send a **resizeText::** message directly, but you might override it. *oldBounds* can differ from **bounds** in **origin.x** and **size.width** and **size.height**. Returns **self.**

scanFunc

– (NXTextFunc)**scanFunc**

Returns the scan function, the function that calculates the contents of each line of text given the line width, font size, text alignment, and other factors. **NXScanALine()** is the default scan function.

See also: – **setScanFunc:**, – **setDrawFunc:**

scrollSelToVisible

– **scrollSelToVisible**

Scrolls the text so that the selection is visible. Returns **self**.

selectAll:

– **selectAll:***sender*

Attempts to make a Text object the first responder and, if successful, then selects all of its text. Returns **self**.

See also: – **selectError**, – **setSel::**

selectError

– **selectError**

Makes the entire text the selection and highlights it. The Text object applies this method if the delegate requires the Text object to maintain its status as the first responder. You rarely need to send a **selectError** message directly, although you may want to override it. To highlight a portion of the text, use **setSel::**. Returns **self**.

See also: – **setSel::**, – **setDelegate:**, – **selectAll:**

selectNull

– **selectNull**

Removes the selection and makes the highlighting (or caret, if the selection is zero-length) disappear. The Text object's delegate isn't notified of the change. The Text object sends a **selectNull** message whenever it needs to end the current selection but retain its status as the first responder; you rarely need to override this method or send **selectNull** messages directly. Returns **self**.

See also: – **setSel::**, – **selectError**, – **selectAll:**, – **getSel::**

selectText:

– **selectText:***sender*

Attempts to make a Text object the first responder and, if successful, then selects all of its text. Returns **self**.

See also: – **selectAll:**, – **setSel::**

selGray

– (float)**selGray**

Not yet implemented.

See also: – **setSelGray:**, – **setBackgroundGray:**, – **backgroundGray**, – **setTextGray:**, – **textGray**

setAlignment:

– **setAlignment:**(int)*mode*

Sets the default alignment for the text. *mode* can have these values (NX_LEFTALIGNED is the default):

Constant	**Alignment**
NX_LEFTALIGNED	Flush to left edge of the **bodyRect**.
NX_RIGHTALIGNED	Flush to right edge of the **bodyRect**.
NX_CENTERED	Each line centered between left and right edges of the **bodyRect**.
NX_JUSTIFIED	Flush to left and right edges of the **bodyRect**; justified. Not yet implemented.

setAlignment: doesn't rewrap or redraw the text. Send a **calcLine** message if you want the text rewrapped and redrawn after you reset the alignment. Returns **self**.

See also: – **alignment**, – **calcLine**, – **alignSelLeft:**, – **alignSelCenter:**, – **alignSelRight:**

setBackgroundColor:

– **setBackgroundColor:**(NXColor)*color*

Sets *color* as the background color for the Text object. *color* is an NXColor structure as defined in **appkit/color.h**. If the Text object's window and screen allow it, this color is displayed the next time the text is redrawn. A **setBackgroundColor:** message doesn't cause the text to be redrawn. Returns **self**.

See also: – **setBackgroundGray:**, – **backgroundGray:**, – **backgroundColor**, – **setTextGray:**, – **textGray**, – **setTextColor:**, – **textColor**, – **setSelGray:**, – **selGray**, – **setSelColor:**

setBackgroundGray:

– **setBackgroundGray:**(float)*value*

Sets the gray value for the background of the text. *value* should lie in the range from 0.0 (indicating black) to 1.0 (indicating white). To specify one of the four pure shades of gray, use one of these constants:

Constant	**Shade**
NX_WHITE	White
NX_LTGRAY	Light gray
NX_DKGRAY	Dark gray
NX_BLACK	Black

A **setBackgroundGray:** message doesn't cause the text to be redrawn. Returns **self**.

See also: – **backgroundGray:**, – **setBackgroundColor:**, – **backgroundColor**, –**setTextGray:**, – **textGray**, – **setTextColor:**, – **textColor**, – **setSelGray:**, – **selGray**, – **setSelColor:**

setBreakTable:

– **setBreakTable:**(const NXFSM *)*aTable*

Sets the break table, the finite-state machine table that the Text object uses to determine word boundaries. Returns **self**.

See also: – **breakTable**

setCharCategoryTable:

– **setCharCategoryTable:**(const unsigned char *)*aTable*

Sets the character category table, the table that maps ASCII characters to character categories. Returns **self**.

See also: – **charCategoryTable**

setCharFilter:

– **setCharFilter:**(NXCharFilterFunc)*aFunc*

Sets the character filter function, the function that analyzes each character the user enters. The Text object has two character filter functions: **NXFieldFilter**() and **NXEditorFilter**(). **NXFieldFilter**() interprets Tab and Return characters as commands to end the Text object's status as the first responder. **NXEditorFilter**(), the default filter function, accepts Tab and Return characters into the text. Returns **self**.

See also: – **charFilter**

setCharWrap:

– **setCharWrap:**(BOOL)*flag*

Sets how words whose length exceeds the line length should be treated. If YES, long words are wrapped on a character basis. If NO, long words are truncated at the boundary of the **bodyRect**. Returns **self**.

See also: – **charWrap**

setClickTable:

– **setClickTable:**(const NXFSM *)*aTable*

Sets the finite-state machine table that defines word boundaries for double-click selection. Returns **self**.

See also: – **clickTable**

setDelegate:

– **setDelegate:***anObject*

Sets the Text object's delegate. In response to user input, the Text object can send the delegate any of several notification messages. See the introduction to this class specification for more information. Returns **self**.

See also: – **delegate**, – **acceptsFirstResponder**, – **resignFirstResponder**

setDescentLine:

– **setDescentLine:**(NXCoord)*value*

Sets the default descent line for the text. The descent line is the distance from the bottom of a line of text to the base line of the text. **setDescentLine:** neither rewraps nor redraws the text. Send a **calcLine** message if you want the text rewrapped and redrawn after you reset the descent line. Returns **self**.

See also: – **descentLine**, – **calcLine**

setDrawFunc:

– **setDrawFunc:**(NXTextFunc)*aFunc*

Sets the draw function, the function that's called to draw each line of text. **NXDrawALine()** is the default draw function. Returns **self**.

See also: – **drawFunc**, – **setScanFunc:**

setEditable:

– **setEditable:**(BOOL)*flag*

Sets whether the text can be edited. If *flag* is YES, the text is editable; if NO, the text is read-only. By default, text is editable.

Use **setEditable:** if you don't expect the text's edit status to change. If your application needs to change the text's edit status repeatedly, have the text's delegate implement the appropriate notification methods (see **setDelegate:**). Returns **self**.

See also: – **isEditable**, – **setDelegate:**

setFont:

– **setFont:***fontObj*

Sets the font for the entire text. The entire text is then rewrapped and redrawn. Returns **self**.

See also: – **setFont:paraStyle:**, – **setSelFont:**

setFont:paraStyle:

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

Sets the font and paragraph style for the entire text. The text is then rewrapped and redrawn. The paragraph style controls such features as tab stops and line indentation. Returns **self**.

See also: – **setFont:**, – **setSelFont:**, – **setParaStyle:**

setFontPanelEnabled:

– **setFontPanelEnabled:**(BOOL)*flag*

This sets whether the Text object will respond to the **changeFont:** message issued by the Font panel. If enabled, the Text object will allow the user to change the font of the selection for a rich Text object. For a plain Text object, the font for the entire text is changed. If enabled, the Text object also updates the Font panel's font selection information. Returns **self**.

See also: – **isFontPanelEnabled**

setHorizResizable:

– **setHorizResizable:**(BOOL)*flag*

Sets whether the text can change size horizontally. If flag is YES, the Text object's frame rectangle can change in the horizontal dimension in response to additions or deletions of text; if NO, it can't. By default, the Text object can't change size. Returns **self**.

See also: – **setVertResizable:**, – **isVertResizable**, – **isHorizResizable**

setLineHeight:

– **setLineHeight:**(NXCoord)*value*

Sets the default minimum distance between adjacent lines. For a plain Text object, this will be the same for all lines. For rich Text objects, line heights will be increased for lines with larger fonts. Even if very small fonts are used, in no case will adjacent lines be closer than this minimum. **setLineHeight:** neither rewraps nor redraws the text. Send a **calcLine** message if you want the text rewrapped and redrawn after you reset the line height. If no line height is set, the default line height will be taken from the default font. Returns **self**.

See also: – **lineHeight**, + **setDefaultFont:**, – **calcLine**

setLocation:ofCell:

– **setLocation:**(NXPoint *)*origin* **ofCell:***cell*

Sets the x and y coordinates for the Cell object specified by *cell*. The coordinates are contained in the structure referred to by *origin* and are interpreted as being in the Text object's coordinate system.

This method is provided for programmers who want to write their own scan functions and need a way to position Cell objects found in the text stream. Sending a **setLocation:ofCell:** message to a Text object that uses the standard scan function will have no effect on the placement of *cell*. Returns **self**.

See also: – **getLocation:ofCell:**, – **replaceSelWithCell:**

setMarginLeft:right:top:bottom:

– **setMarginLeft:**(NXCoord)*leftMargin*
  right:(NXCoord)*rightMargin*
  top:(NXCoord)*topMargin*
  bottom:(NXCoord)*bottomMargin*

Adjusts the dimensions of the Text object's margins. Returns **self**.

See also: – **getMarginLeft:right:top:bottom:**

setMaxSize:

– **setMaxSize:**(const NXSize *)*newMaxSize*

Sets the maximum size of a Text object. This maximum size is ignored if the Text object can't be resized. The default maximum size is {0.0, 0.0}. Returns **self**.

See also: – **getMaxSize:**, – **setMinSize:**

setMinSize:

– **setMinSize:**(const NXSize *)*newMinSize*

Sets the minimum size of the receiving Text object. This size is ignored if the Text object can't be resized. The default minimum size is {0.0, 0.0}. Returns **self**.

See also: – **getMinSize:**, – **setMaxSize:**

setMonoFont:

– **setMonoFont:**(BOOL)*flag*

Sets whether the receiving Text object uses one font and paragraph style for the entire text. By default, a Text object allows only one font and paragraph style. Messages to set the font, line height, text alignment, and so on affect the entire text of such Text objects. Text pasted into such Text objects assume their current font and alignment characteristics. A Text object in this state is called a plain Text object.

By sending a **setMonoFont:**NO message, multiple fonts and paragraph styles can be displayed in a Text object. Thereafter, font changes affect only the selected text, and paragraph style changes affect only the paragraph or paragraphs marked by the selection. The font and alignment characteristics of pasted text are maintained. A Text object in this state is called a rich Text object. Returns **self**.

See also: – **isMonoFont**, – **alignSelLeft:**, – **setSelProp:to:**, – **setFontPanelEnabled:**

setNoWrap

– **setNoWrap**

Sets the Text object's **breakTable** and **charWrap** instance variables so that word wrap is disabled. It also sets the text alignment to NX_LEFTALIGNED. Returns **self**.

See also: – **setCharWrap:**

setOverstrikeDiacriticals:

– **setOverstrikeDiacriticals:**(BOOL)*flag*

Unimplemented.

setParaStyle:

– **setParaStyle:**(void *)*paraStyle*

Sets the paragraph style for the entire text. The text is then rewrapped and redrawn. The paragraph style controls features such as tab stops and line indentation. Returns **self**.

See also: – **setFont:**, – **setFont:paraStyle:**, – **setSelFont:**

setPostSelSmartTable:

– **setPostSelSmartTable:**(const unsigned char *)*aTable*

Sets **postSelSmartTable**, the table that specifies which characters on the right end of a selection are treated as equivalent to a space character. Returns **self**.

See also: – **postSelSmartTable**, – **setPreSelSmartTable:**, – **preSelSmartTable**

setPreSelSmartTable:

– **setPreSelSmartTable:**(const unsigned char *)*aTable*

Sets **preSelSmartTable**, the table that specifies which characters on the left end of a selection are treated as equivalent to a space character. Returns **self**.

See also: – **preSelSmartTable**, – **setPostSelSmartTable:**

setRetainedWhileDrawing:

– **setRetainedWhileDrawing:**(BOOL)*flag*

Sets whether the Text object automatically changes its window's buffering type from buffered to retained whenever it redraws itself. Drawing directly to the screen improves the Text object's perceived performance, especially if the text contains numerous fonts and formats. Rather than waiting until the entire text is flushed to the screen, the user sees the text being drawn line-by-line.

The window's buffering type changes to retained only while the Text object is redrawing itself—that is, only when the Text object's **drawSelf::** method is invoked. In other cases, such as when a user is entering text, the window's buffering type is unaffected. This method is designed to work with Text objects that are in buffered windows; don't send a **setRetainedWhileDrawing:** message to a Text object in a retained or nonretained window. Returns **self**.

See also: – **isRetainedWhileDrawing**, – **drawSelf::**

setScanFunc:

– **setScanFunc:**(NXTextFunc)*aFunc*

Sets the scan function, the function that calculates the contents of each line of text given the line width, font size, type of text alignment, and other factors. **NXScanALine()** is the default scan function. Returns **self**.

See also: – **scanFunc**, – **setDrawFunc:**

setSel::

– **setSel:**(int)*start* **:**(int)*end*

Makes the Text object the first responder and then selects and highlights a portion of the text. *start* is the first character position of the selection; *end* is the last character position of the selection. To create an empty selection, *start* must equal *end*. Use **setSel::** to select a portion of the text programmatically. Returns **self**.

See also: – **selectAll:**, – **selectError**, – **selectNull**, – **getSel::**

setSelColor:

– **setSelColor:**(NXColor)*color*

Sets the text color of the selected text, assuming the Text object allows more than one paragraph style and font (see **setMonoFont:**). Otherwise, **setSelColor:** sets the text color for the entire text. *color* is an NXColor structure as defined in the header file **appkit/color.h**. After the text color is set, the text is redisplayed. Returns **self**.

See also: – **setBackgroundGray:**, – **backgroundGray:**, – **setBackgroundColor:**, – **backgroundColor**, – **setTextGray:**, – **textGray**, – **setTextColor:**, – **textColor**, – **setSelGray:**, – **selGray**

setSelectable:

– **setSelectable:**(BOOL)*flag*

Sets whether the text can be selected. By default, text is selectable. Returns **self**.

See also: – **isSelectable**, – **setEditable:**

setSelFont:

– **setSelFont:***fontId*

Sets the font for the selection. The text is then rewrapped and redrawn. Returns **self.**

See also: – **setSelFontSize:**, – **setSelFontStyle:**, – **setFont:**

setSelFont:paraStyle:

– **setSelFont:***fontId* **paraStyle:**(void *)*paraStyle*

Sets the font of the current selection to that specified by *fontID*. The paragraph style is also changed. Returns **self**.

See also: – **setSelFont:**, – **setSelFontSize:**, – **setSelFontStyle:**

setSelFontFamily:

– **setSelFontFamily:**(const char *)*fontName*

Sets the name of the font for the selection to *fontName*. The text is then rewrapped and redrawn. Returns **self**.

See also: – **setSelFontSize:**, – **setSelFontStyle:**

setSelFontSize:

– **setSelFontSize:**(float)*size*

Sets the size of the font for the selection to *size*. The text is then rewrapped and redrawn. Returns **self**.

See also: – **setSelFont:**, – **setSelFontStyle:**, – **setFont:**

setSelFontStyle:

– **setSelFontStyle:**(NXFontTraitMask)*traits*

Sets the font style for the selection. The text is then rewrapped and redrawn. The Text object uses the FontManager to change the various traits of the selected font. Returns **self**.

See also: – **setSelFont:**, – **setSelFontSize:**, – **setFont:**

setSelGray:

– **setSelGray:**(float)*value*

Sets the gray value of the selected text, assuming the Text object allows more than one paragraph style and font (see **setMonoFont:**). Otherwise, **setSelGray:** sets the gray value for the entire text. *value* should lie in the range 0.0 (indicating black) to 1.0 (indicating white). To specify one of the four pure shades of gray, use one of these constants:

Constant	**Shade**
NX_WHITE	White
NX_LTGRAY	Light gray
NX_DKGRAY	Dark gray
NX_BLACK	Black

After the gray value is set, the text is redisplayed. Returns **self**.

See also: – **setBackgroundGray:**, – **backgroundGray:**, – **setBackgroundColor:**, – **backgroundColor**, – **setTextGray:**, – **textGray**, – **setTextColor:**, – **textColor**, – **selGray**, – **setSelColor:**

setSelProp:to:

– **setSelProp:**(NXParagraphProp)*prop* **to:**(NXCoord)*val*

Sets the paragraph style for one or more paragraphs. For a plain Text object, all paragraphs are affected. For a rich Text object, only those paragraphs marked by the selection are affected. *prop* determines which property is modified, and *val* provides additional information needed for some properties. These constants are defined for *prop*:

Constant	**Property Affected**
NX_LEFTALIGN	Text alignment. Aligns the text to the left margin. *val* is ignored.
NX_RIGHTALIGN	Text alignment. Aligns the text to the right margin. *val* is ignored.
NX_CENTERALIGN	Text alignment. Centers the text between the left and right margins. *val* is ignored.
NX_JUSTALIGN	Not yet implemented.
NX_FIRSTINDENT	Indentation of the first line. *val* specifies the number of units (in the receiver's coordinate system) along the x axis to indent.
NX_INDENT	Indentation of lines other than the first line. *val* specifies the number of units (in the receiver's coordinate system) along the x axis to indent.
NX_ADDTAB	Tab placement. *val* specifies the position on the x axis (in the receiver's coordinate system) to add the new tab.
NX_REMOVETAB	Tab placement. *val* identifies the tab to be removed by specifying its position on the x axis (in the receiver's coordinate system).
NX_LEFTMARGIN	Left margin width. *val* gives the new width as a number of units in the receiver's coordinate system.
NX_RIGHTMARGIN	Right margin width. *val* gives the new width as a number of units in the receiver's coordinate system.

setSelProp:to: sets the left and right margins by performing the **setMarginLeft:right:top:bottom:** method. For all other properties, it performs the **setFont:parastyle:** method. After the paragraph property is set, the text is rewrapped and redrawn. Returns **self**.

See also: – **alignSelCenter:**, – **alignSelLeft:**, – **alignSelRight:**, – **setMonoFont:**

setTag:

– **setTag:**(int)*anInt*

Sets the Text object's **tag** value to *anInt*. Returns **self**.

See also: – **tag**, – **findViewWithTag:**

setText:

– **setText:**(const char *)*aString*

Replaces the current text with the text referred to by *aString*. The Text object then wraps and redraws the text, if autodisplay is enabled. This method doesn't affect the object's frame or bounds rectangle. To resize the text rectangle to make the text entirely visible, use the **sizeToFit** method. Returns **self**.

See also: – **setSel::**, – **readText:**, – **readRichText:**, – **sizeToFit**

setTextColor:

– **setTextColor:**(NXColor)*color*

Sets *color* as the text color for the entire text. *color* is an NXColor structure as defined in the header file **appkit/color.h**. If the Text object's window and screen allow it, this color is displayed the next time the text is redrawn. **setTextColor:** doesn't redraw the text. Returns **self**.

To set the color of selected text, use **setSelColor:**.

See also: – **setBackgroundGray:**, – **backgroundGray:**, – **setBackgroundColor:**, – **backgroundColor**, – **setTextGray:**, – **textGray**, – **textColor**, – **setSelGray:**, – **selGray**, – **setSelColor:**

setTextFilter:

– **setTextFilter:**(NXTextFilterFunc)*aFunc*

Sets the text filter function, the function that analyzes text the user enters.

The text filter function is called with the following arguments:

```
NXTextFunc myTextFilter(id self, unsigned char *insertText,
                         int *insertLength, int position);
```

This function may change the contents of the text to be inserted. The pointer to the new text is returned, and the new length is written into the *insertLength* integer pointer. The position is where the new text is to be inserted.

This filter is different from the character filter in that you're given where the text is to be inserted and the new text that will be inserted. This enables you to write a filter to do auto-indent, or a filter to allow only properly formatted floating point numbers. The character filter doesn't give enough context to determine exactly what the state of the Text object is before and after the edit. Returns **self**.

See also: – **textFilter**

setTextGray:

– **setTextGray:**(float)*value*

Sets the gray value for the entire text. *value* should lie in the range 0.0 (indicating black) to 1.0 (indicating white). To specify one of the four pure shades of gray, use one of these constants:

Constant	**Shade**
NX_WHITE	White
NX_LTGRAY	Light gray
NX_DKGRAY	Dark gray
NX_BLACK	Black

A **setTextGray:** message doesn't cause the text to be redrawn. Returns **self**.

See also: – **setBackgroundGray:**, – **backgroundGray:**, – **setBackgroundColor:**, – **backgroundColor**, – **textGray**, – **setTextColor:**, – **textColor**, – **setSelGray:**, – **selGray**, – **setSelColor:**

setVertResizable:

– **setVertResizable:**(BOOL)*flag*

Sets whether the text can change size vertically. If flag is YES, the Text object's frame rectangle can change in the vertical dimension in response to additions or deletions of text; if NO, it can't. By default, a Text object can't change size. Returns **self.**

See also: – **isVertResizable**, – **setHorizResizable:**, – **isHorizResizable**

showCaret

– **showCaret**

Displays the caret. The Text object sends itself **showCaret** messages whenever it needs to redisplay the caret; you rarely need to send a **showCaret** message directly. Occasions when the **showCaret** message is sent include whenever a Text object receives **becomeKeyWindow**, **paste:**, or **delete:** messages. A **showCaret** message redisplays the caret only if the selection is zero-width. If the Text object is not in a window, or the window is not the key window, or the Text object is not editable, this method has no effect. Returns **self.**

See also: – **hideCaret**

showGuessPanel:

– **showGuessPanel:***sender*

Displays a panel that offers suggested alternate spellings for a word that's suspected of being misspelled. The user can either accept one of the alternates, added the word to a local dictionary in **~/.NeXT/LocalDictionary**, or skip the word.

A word becomes a candidate for the Guess panel's actions by being selected as the result of the Text object's receiving a **checkSpelling:** message. Returns **self**.

See also: – **checkSpelling:**

sizeTo::

– **sizeTo:**(NXCoord)*width* **:**(NXCoord)*height*

Sets the Text object's frame rectangle to the specified width and height in its superview's coordinates. This method doesn't rewrap the text; to do that, send a **calcLine** message. Returns **self**.

See also: – **sizeTo::** (View)

sizeToFit

– **sizeToFit**

Modifies the frame rectangle to completely display the text. This is often used with Text objects in a ScrollView. The **setHorizResizable:** and **setVertResizable:** methods determine whether the Text object will resize horizontally or vertically (by default, it won't change size in either dimension). After receiving a **calcLine** message, a Text that is the document view of a ScrollView sends itself a **sizeToFit** message. See **calcLine** for the methods that send **calcLine** messages. Returns **self**.

See also: – **setHorizResizable:**, – **setVertResizable:**

startReadingRichText

– **startReadingRichText**

A **startReadingRichText** message is sent to the Text object just before it begins reading RTF data. The Text object responds by sending its delegate a **textWillStartReadingRichText:** message, assuming there is a delegate and it responds to this message. The delegate can then perform any required initialization. Alternatively, a subclass of Text could put these initialization routines in its own implementation of this method. Returns **self**.

stream

– (NXStream *)**stream**

Returns a pointer to a read-only stream that allows you to read the contents of the Text object. The returned stream is convenient for parsing the contents of the Text object or for implementing text searching within a text editor. The stream is valid until the Text object is edited. You shouldn't keep a copy of the stream (or free the stream) after you finish using it. When you need the stream again, send another **stream** message to get a valid one.

See also: – **getSubstring:start:length:**, – **firstTextBlock**, – **stream**

subscript:

– **subscript:***sender*

Subscripts the selection. The text is then rewrapped and redrawn. The text is subscripted by 40% of the selection's font height. Returns **self**.

See also: – **superscript:**, – **unscript:**

superscript:

– **superscript:***sender*

Superscripts the selection. The text is then rewrapped and redrawn. The text is superscripted by 40% of the selection's font height. Returns **self**.

See also: – **subscript:**, – **unscript:**

tag

– (int)**tag**

Returns the Text object's tag.

See also: – **setTag:**, – **findViewWithTag:**

textColor

– (NXColor)**textColor**

Returns an NXColor structure that denotes the color used for drawing text.

See also: – **setTextColor:**

textFilter

– (NXTextFilterFunc)**textFilter**

Returns the text filter function, the function that analyzes text the user enters. By default, this function is NULL.

See also: – **setTextFilter:**

textGray

– (float)**textGray**

Returns the gray value used to draw the text.

See also: – **setTextGray:**

textLength

– (int)**textLength**

Returns the number of characters in a Text object. The length doesn't include the null terminator ('\0') that **getSubstring:start:length:** returns if you ask for all the text in a Text object.

See also: – **byteLength**, – **getSubstring:start:length:**

toggleRuler:

– **toggleRuler:***sender*

Controls the display of the ruler. This method has effect only if the receiving Text object is a subview of a ScrollView. **toggleRuler:** causes the ScrollView to display a ruler if one isn't already present, or to remove the ruler if one is. When the ruler is displayed, its settings reflect the paragraph style of the paragraph containing the selection.

sender is the **id** of the sending object. Returns **nil** if the receiver isn't a subview of a ScrollView instance; otherwise, returns **self**.

See also: – **isRulerVisible:**, – **copyRuler:**, – **pasteRuler:**

underline:

– **underline:***sender*

Toggles the underline attribute of text. This method has effect only if the receiving Text object can display multiple fonts and paragraph styles (see **setMonoFont:**).

underline: adds an underline to the selected text if one doesn't already exist or removes the underline if it does. If the selection is zero-width, **underline:** affects the underline attribute of text that's subsequently entered at the insertion point.

sender is the **id** of the sending object. Returns **self**.

See also: – **setMonoFont:**, – **superscript:**, – **subscript:**

unscript:

– **unscript:***sender*

Removes the subscript or superscript property of the current selection. The text is then rewrapped and redrawn. Returns **self**.

See also: – **subscript:**, – **superscript:**

validRequestorForSendType:andReturnType:

– **validRequestorForSendType:**(NXAtom)*sendType*
  andReturnType:(NXAtom)*returnType*

Responds to a message that the Application object sends to determine which items in the Services menu should be enabled or disabled at any particular time. You never send a **validRequestorForSendType:andReturnType:** message directly, but you might override this method in a subclass of Text.

A Text object registers for services during initialization (however, see **excludeFromServicesMenu:**). Thereafter, whenever the Text object is the first responder, the Application object can send it one or more **validRequestorForSendType:andReturnType:** messages during event processing to determine which Services menu items should be enabled. If the Text object can place data of type *sendType* on the pasteboard and receive data of type *returnType* back, it should return **self**; otherwise it should return **nil**. The Application object checks the return value to determine whether to enable or disable commands in the Services menu.

Since an object can receive one or more of these messages per event, it's important that if you override this method in a subclass of Text, the new implementation include no time-consuming calculations.

See the description of **validRequestorForSendType:andReturnType:** in the Responder class specification for more information.

See also: + **excludeFromServicesMenu:**,
– **registerServicesMenuSendTypes:andReturnTypes:** (Application),
– **readSelectionFromPasteboard:**, – **writeSelectionToPasteboard:**,
– **validRequestorForSendType:andReturnType:** (Responder)

windowChanged:

– **windowChanged:***newWindow*

Notifies the receiving Text object of a change in the identity of its Window. Generally, the change is the result of the Text object (or one of its superviews) being removed from the Window's view hierarchy. This method ensures that the caret is hidden whenever the window changes. Returns **self**.

See also: – **windowChanged:** (View)

write:

– **write:**(NXTypedStream *)*stream*

Writes the Text object to the typed stream *stream*. A **write:** message is sent in response to archiving; you never send this message directly. Returns **self**.

writeRichText:

– **writeRichText:**(NXStream *)*stream*

Writes the contents of the Text object as RTF data to *stream*. The margins, fonts, superscripting/subscripting, text color, and text are written out in this format. See the *NextStep Concepts* manual for the subset of RTF directives that's supported. Returns **self**.

See also: – **writeText:**, – **readText:**

writeRichText:forRun:atPosition:emitDefaultRichText:

– **writeRichText:**(NXStream *)*stream*
  forRun:(NXRun *)*run*
  atPosition:(int)*runPosition*
  emitDefaultRichText:(BOOL *)*writeDefaultRTF*

You never send this message, but may want to override it to write special RTF directives while the Text object is writing RTF data. Returns **self**.

writeRichText:from:to:

– **writeRichText:**(NXStream *)*stream*
    from:(int)*start*
    to:(int)*end*

Writes a portion of the text starting at position *start* to position *end* in RTF to *stream*. Returns **self**.

See also: – **writeText:**, – **readText:**

writeSelectionToPasteboard:types:

– (BOOL)**writeSelectionToPasteboard:***pboard*
    types:(NXAtom *)*types*

Writes the current selection to the supplied Pasteboard object, *pboard*. *types* lists the data types to be copied to the pasteboard. A return value of NO indicates that the data of the requested types could not be provided.

When the user chooses a command in the Services menu, a **writeSelectionToPasteboard:types:** message is sent to the first responder. This message is followed by a **readSelectionFromPasteboard:** message if the command requires the requesting application to replace its selection with data from the service provider.

See also: – **readSelectionFromPasteboard:**,
– **validRequestorForSendType:andReturnType:**

writeText:

– **writeText:**(NXStream *)*stream*

Writes the entire text to *stream*. If you want to write only the selected text to a stream, use **getSel::** (to determine the extent of the selection), **getSubstring:start:length:** (to retrieve the text within the selected region), and then **NXWrite()** to write the text to the stream. Returns **self**.

See also: – **writeRichText:**, – **readText:**, – **getSubstring:start:length:**

METHODS IMPLEMENTED BY THE DELEGATE

textDidChange:

– **textDidChange:***sender*

Responds to a message sent to the delegate after the first change to the text since the Text object became the first responder. The delegate receives a **textWillChange:** message immediately before receiving a **textDidChange:** message.

textDidEnd:endChar:

– **textDidEnd:***sender* **endChar:**(unsigned short)*whyEnd*

Responds to a message informing the delegate that the Text object has relinquished first responder status. *whyEnd* is the movement character (Tab, Shift-Tab, or Return) that caused the Text object to cease being the first responder. The delegate can use this information to decide which other object should become the first responder.

textDidGetKeys:isEmpty:

– **textDidGetKeys:***sender* **isEmpty:**(BOOL)*flag*

Responds to a message sent to the delegate after each change to the text. *flag* indicates whether the Text object contains any text after the change.

textDidRead:paperSize:

– **textDidRead:***sender* **paperSize:**(NXSize *)*paperSize*

Responds to a message informing the delegate that the Text object will read the paper size for the document.

This message is sent to the delegate after the Text object reads RTF data, allowing the delegate to modify the paper size. *paperSize* is the dimensions of the paper size specified by the \ **paperw** and \ **paperh** RTF directives.

See also: – **textWillWrite:paperSize:**

textDidResize:oldBounds:invalid:

– **textDidResize:***sender*
  oldBounds:(const NXRect *)*oldBounds*
  invalid:(NXRect *)*invalidRect*

Responds to a message informing the delegate that the Text object has changed its size. *oldBounds* is the Text object's bounds rectangle before the change. *invalidRect* is the area of the Text object's superview that should be redrawn if the Text object has become smaller.

textWillChange:

– (BOOL)**textWillChange:***sender*

Responds to a message sent upon the first user input since the Text object became the first responder. The delegate's **textWillChange:** method can prevent the text from being changed by returning a nonzero value. If the delegate allows the change, it immediately receives a **textDidChange:** message after the change is made. If the delegate doesn't implement this method, the change is allowed by default.

textWillConvert:fromFont:toFont:

– **textWillConvert:***sender*
  fromFont:*from*
  toFont:*to*

Responds to a message giving the delegate the opportunity to alter the font that will be used for the selection. The message is sent whenever the Font panel sends a **changeFont:** message to the Text object. *from* is the old font that's currently being changed, *to* is the font that's to replace *from*. This method returns the font that's to be used instead of the *to* font.

textWillEnd:

– (BOOL)**textWillEnd:***sender*

Responds to a message informing the delegate that the Text object is about to relinquish first responder status. The delegate's **textWillEnd:** method can prevent the change by returning a nonzero value. If the delegate prevents the change, the entire text becomes selected. If the delegate doesn't implement this method, the change is allowed by default.

textWillFinishReadingRichText:

– **textWillFinishReadingRichText:***sender*

Responds to a message informing the delegate that the Text object has read RTF data, either from the pasteboard or from a text file.

textWillReadRichText:stream:atPosition:

– **textWillReadRichText:***sender*
  stream:(NXStream *)*stream*
  atPosition:(int)*runPosition*

This method is the inverse operation from **textWillWriteRichText:stream:forRun:atPosition:emitDefaultRichText:**. This method must read the same number of characters from *stream* that the inverse operation emits.

See also:
– textWillWriteRichText:stream:forRun:atPosition:emitDefaultRichText:

textWillResize:

– **textWillResize:***sender*

Responds to a message informing the delegate that the Text object is about to change its size. The delegate's **textWillResize:** method can specify the maximum dimensions of the Text object by using the **setMaxSize:** method.

If the delegate doesn't implement this method, the change is allowed by default.

textWillSetSel:toFont:

– **textWillSetSel:***sender* **toFont:***font*

Responds to a message giving the delegate the opportunity to change the font that the Text object is about to display in the Font panel. *font* is the font that's about to be set in the Font panel. This method returns the real font to show in the Font panel.

textWillStartReadingRichText:

– **textWillStartReadingRichText:***sender*

Responds to a message informing the delegate that the Text object is about to read RTF data, either from the Pasteboard or from a text file.

textWillWrite:paperSize:

– **textWillWrite:***sender* **paperSize:**(NXSize *)*paperSize*

Responds to a message informing the delegate that the Text object will write out the paper size for the document.

As part of its RTF output, theText object's delegate can write out a paper size for the document. The delegate specifies the paper size by placing the width and height values (in points) in the structure referred to by *paperSize*. Unless the delegate specifies otherwise, the paper size is assumed to be 612 by 792 points (8 1/2 by 11 inches).

See also: – **textDidRead:paperSize:**

textWillWriteRichText:stream:forRun:atPosition:emitDefaultRichText:

– **textWillWriteRichText:***sender*
  stream:(NXStream *)*stream*
  forRun:(NXRun *)*run*
  atPosition:(int)*runPosition*
  emitDefaultRichText:(BOOL *)*writeDefaultRichText*

The delegate may choose to write additional information into the RTF output. Runs that have the **rFlags.subclassWantRTF** field set will be sent as *run* in this message. The additional information should be written to *stream*, in an ASCII format. The **textWillReadRichText:stream:atPosition:** method, which does the inverse operation when RTF data is read, must read the same number of characters as is written by **textWillWriteRichText:stream:forRun:atPosition:emitDefaultRichText:**. *runPosition* is the position in the text stream that *run* describes; the length of the run is in the **chars** field of the NXRun structure. If YES, *writeDefaultRichText* instructs the Text object to write out the normal RTF data for the run *run*.

See also: – **textWillReadRichText:stream:atPosition:**

METHODS IMPLEMENTED BY AN EMBEDDED GRAPHIC OBJECT

calcCellSize:

– **calcCellSize:**(NXSize *)*theSize*

Responds to a message from the Text object by providing the graphic object's width and height. The Text object uses this information to adjust character placement and line height to accommodate the display of the graphic object in the text. See the Cell class specification for one implementation of this method.

See also: – **calcCellSize:** (Cell)

drawSelf:inView:

– **drawSelf:**(const NXRect *)*rect* **inView:***view*

Responds to a message from the Text object by drawing the graphic object within the given rectangle and View. The supplied View is generally the Text object itself. See the Cell class specification for one implementation of this method.

See also: – **drawSelf:inView:** (Cell)

highlight:inView:lit:

– **highlight:**(const NXRect *)*rect* **inView:***view* **lit:**(BOOL)*flag*

Responds to a message from the Text object by highlighting or unhighlighting the graphic object during mouse tracking. *rect* is the area within *view* (generally the Text object itself) to be highlighted. If *flag* is YES, this method should draw the graphic object in its highlighted state; if NO, it should draw the graphic object in its normal state. See the Cell class specification for one implementation of this method.

See also: – **highlight:inView:lit:** (Cell)

readRichText:forView:

– **readRichText:**(NXStream *)*stream* **forView:***view*

Responds to a message sent by the Text object when it encounters an RTF control word that's associated with the graphic object's class (see **registerDirective:forClass:**). The graphic object should read its representation from the RTF data in the supplied stream. The Text object passes its **id** as the *view* argument.

This method is the counterpart to **writeRichText:forView:**. In extracting the image data from the stream, **readRichText:forView:** must read the exact number of characters that **writeRichText:forView:** wrote in storing the image data to the stream.

See also: – **writeRichText:forView:**, – **registerDirective:forClass:**

trackMouse:inRect:ofView:

– (BOOL)**trackMouse:**(NXEvent *)*theEvent* **inRect:**(const NXRect *)*rect*
ofView:*view*

Responds to a message from the Text object by tracking the mouse while it's within the specified rectangle of the supplied View. *theEvent* is a pointer to the mouse-down event that caused the Text object to send this message. *rect* is the area within *view* (generally the Text object) where the mouse will be tracked. See the Cell class specification for one implementation of this method.

See also: – **trackMouse:inRect:ofView:** (Cell)

writeRichText:forView:

– **writeRichText:**(NXStream *)*stream* **forView:***view*

Responds to a message sent by the Text object when it encounters the graphic object in the text it's writing to *stream.* The graphic object should write an RTF representation of its image to the supplied stream. The Text object passes its **id** as the *view* argument.

See also: – **readRichText:forView:**, – **registerDirective:forClass:**

CONSTANTS AND DEFINED TYPES

```
#define NX_TEXTPER  490         /* Number of characters to allocate */
                                /* for each text block */

typedef struct _NXTextBlock {
    struct _NXTextBlock *next;  /* Next block in linked list */
    struct _NXTextBlock *prior; /* Previous block in linked list */
    struct _tbFlags {
       unsigned int      malloced:1; /* True if block was malloced */
       unsigned int      PAD:15;
    } tbFlags;
    short                chars; /* Number of characters in block */
    unsigned char       *text;  /* The text */
} NXTextBlock;

typedef struct {
    unsigned int underline:1;         /* True if text is underlined */
    unsigned int dummy:1;             /* Unused */
    unsigned int subclassWantsRTF:1;  /* Obsolete */
    unsigned int graphic:1;           /* True if graphic is present */
    unsigned int RESERVED:12;
} NXRunFlags;

/* NXRun represents a single sequence of text with a given format. */
typedef struct _NXRun {
    id             font;           /* Font id */
    int            chars;          /* Number of characters in run */
    void          *paraStyle;      /* Implementation-dependent */
                                   /* paragraph style information */
    float          textGray;       /* Text gray of current run */
    float          textRGBColor;   /* Text color of current run */
    unsigned char  superscript;    /* Superscript in points */
    unsigned char  subscript;      /* Subscript in points */
    id             info;           /* For subclasses of Text */
    NXRunFlags     rFlags;         /* Indicates underline etc. */
} NXRun;

/* An NXRunArray holds the array of text runs.*/
typedef struct _NXRunArray {
    NXChunk   chunk;
    NXRun     runs[1];
} NXRunArray;
```

```
/*
 * An NXBreakArray holds line break information.  It's mainly an
 * array of line descriptors.  Each line descriptor contains three
 * fields:
 *
 *    1) Line change bit (sign bit); set if this line defines a new
 *       height
 *    2) Paragraph end bit (next to sign bit); set if the end of this
 *       line ends the paragraph
 *    3) Number of characters in the line (low-order 14 bits).
 *
 * If the line change bit is set, the descriptor is the first field
 * of an NXHeightChange structure.  Since this record is bracketed
 * by negative short values, the breaks array can be sequentially
 * accessed backwards and forwards.
 */

typedef short NXLineDesc;        /* Line descriptor */

typedef struct _NXHeightInfo {
    NXCoord      newHeight;      /* Line height from here forward*/
    NXCoord      oldHeight;      /* Height before change */
    NXLineDesc   lineDesc;       /* Line descriptor */
} NXHeightInfo;

typedef struct _NXHeightChange {
    NXLineDesc   lineDesc;       /* Line descriptor */
    NXHeightInfo heightInfo;
} NXHeightChange;

typedef struct _NXBreakArray {
    NXChunk      chunk;
    NXLineDesc   breaks[1];      /* Line descriptor */
} NXBreakArray;

/*
 * NXLay represents a single sequence of text in a line and records
 * everything needed to select or draw that piece.
 */

typedef struct {
    unsigned int mustMove:1;    /* True if lay follows lay with */
                                /* nonprinting character; e.g. Tab  */
    unsigned int isMoveChar:1;  /* True if lay contains nonprinting */
                                /* character; e.g. Tab */
    unsigned int RESERVED:14;
} NXLayFlags;
```

```
typedef struct _NXLay {
    NXCoord    x;           /* x coordinate of moveto */
    NXCoord    y;           /* y coordinate of moveto */
    short      offset;      /* Offset in line for first character */
                            /* of run */
    short      chars;       /* Number of characters in lay */
    id         font;        /* Font id */
    void      *paraStyle;   /* Implementation-dependent paragraph */
                            /* style information */
    NXRun     *run;         /* Run for lay */
    NXLayFlags lFlags;      /* Indicates lay affected by move */
                            /* characters */
} NXLay;

/* NXLayArray holds the layout for the current line. */
typedef struct _NXLayArray {
    NXChunk       chunk;
    NXLay         lays[1];
} NXLayArray;

/* NXWidthArray holds the widths for the current line. */
typedef struct _NXWidthArray {
    NXChunk       chunk;
    NXCoord       widths[1];
} NXWidthArray;

/* NXCharArray holds the character array for the current line. */
typedef struct _NXCharArray {
    NXChunk       chunk;
    unsigned char text[1];
} NXCharArray;

/*
 * An NXFSM is a word definition finite-state machine transition
 * structure.
 */
typedef struct _NXFSM {
    const struct _NXFSM *next;   /* State to go to; NULL implies
                                    final state */
    short  delta;       /* If final state, this undoes lookahead */
    short  token;       /* If final state, negative value implies */
                        /* word is newline; 0 implies dark; */
                        /* positive implies white space */
} NXFSM;
```

```
/* Represents one end of a selection. */
typedef struct _NXSelPt {
    int          cp;      /* Character position */
    int          line;    /* Offset of NXLineDesc in break table */
    NXCoord      x;       /* x coordinate */
    NXCoord      y;       /* y coordinate */
    int          c1st;    /* Character position of first character */
                          /* on the line */
    NXCoord      ht;      /* Line height */
} NXSelPt;

/* Describes tabstop. */
typedef struct _NXTabStop {
    short        kind;    /* Only NX_LEFTTAB implemented */
    NXCoord      x;       /* x coordinate for stop */
} NXTabStop;

/* Describes current text block and run. */
typedef struct _NXTextCache {
    int          curPos;        /* Current position in text stream */
    NXRun       *curRun;        /* Current run of text */
    int          runFirstPos;   /* Character position of first */
                                /* character in current run */
    NXTextBlock *curBlock;      /* Current block of text */
    int          blockFirstPos; /* Character position of first */
                                /* character in current block */
} NXTextCache;

typedef struct _NXLayInfo {
    NXRect        rect;         /* Bounds rect. for current line. */
    NXCoord       descent;      /* Descent line for current line. */
                                /* Can be reset by scanFunc */
    NXCoord       width;        /* Width of line */
    NXCoord       left;         /* Coordinate visible at left side */
    NXCoord       right;        /* Coord. visible at right side */
    NXCoord       rightIndent;  /* How much white space to leave */
                                /* at right side of line */
    NXLayArray   *lays;         /* Scan function fills with NXLay */
                                /* items */
    NXWidthArray *widths;       /* Scan function fills with  */
                                /* character widths */
    NXCharArray  *chars;        /* Scan function fills with */
                                /* characters */
    NXTextCache   cache;        /* Cache of current block & run */
    NXRect       *textClipRect; /* If not NULL, the current */
                                /* clipping rectangle for drawing */
    struct _lFlags {
        unsigned int horizCanGrow:1;/* True if scan func. should */
                                    /* dynamically resize x margins */
        unsigned int vertCanGrow:1; /* True if scan func. should */
                                    /* dynamically resize y margins */
        unsigned int erase:1;       /* True if draw function should */
                                    /* erase before drawing */
        unsigned int ping:1;        /* True if draw function should */
                                    /* ping Window Server */
        unsigned int endsParagraph:1;/* True if line ends paragraph */
        unsigned int resetCache:1;  /* Used by Scan function to */
                                    /* reset local caches */
        unsigned int RESERVED:10;
    } lFlags;
} NXLayInfo;

/* Describes text layout and tab stops. */
typedef struct _NXTextStyle {
    NXCoord    indent1st;    /* How far first line in paragraph is */
                             /* indented */
    NXCoord    indent2nd;    /* How far second and subsequent lines */
                             /* are indented */
    NXCoord    lineHt;       /* Line height */
    NXCoord    descentLine;  /* Distance from baseline to  */
                             /* bottom of line */
    short      alignment;    /* Text alignment */
    short      numTabs;      /* Number of tab stops */
    NXTabStop *tabs;         /* Array of tab stops */
} NXTextStyle;

/* Text alignment modes. */
#define NX_LEFTALIGNED   0
#define NX_RIGHTALIGNED  1
#define NX_CENTERED      2
#define NX_JUSTIFIED     3

/* Tab stop types. */
#define NX_LEFTTAB       0

/* Constants used by the character filter function. */
#define NX_BACKSPACE      8
#define NX_CR            13
#define NX_DELETE        ((unsigned short)0x7F)
#define NX_BTAB          25
#define NX_ILLEGAL        0
#define NX_RETURN        ((unsigned short)0x10)
#define NX_TAB           ((unsigned short)0x11)
#define NX_BACKTAB       ((unsigned short)0x12)
#define NX_LEFT          ((unsigned short)0x13)
#define NX_RIGHT         ((unsigned short)0x14)
#define NX_UP            ((unsigned short)0x15)
#define NX_DOWN          ((unsigned short)0x16)
```

```
/* Paragraph properties */
typedef enum {
    NX_LEFTALIGN = NX_LEFTALIGNED,
    NX_RIGHTALIGN = NX_RIGHTALIGNED,
    NX_CENTERALIGN = NX_CENTERED,
    NX_JUSTALIGN = NX_JUSTIFIED,
    NX_FIRSTINDENT,
    NX_INDENT,
    NX_ADDTAB,
    NX_REMOVETAB,
    NX_LEFTMARGIN,
    NX_RIGHTMARGIN
} NXParagraphProp;

/*
 * Word tables for various languages.  The SmartLeft and SmartRight
 * arrays are suitable as arguments for the messages
 * setPreSelSmartTable: and setPostSelSmartTable.  When doing a
 * paste, if the character to the left (right) of the new word is not
 * in the left (right) table, an extra space is added on that side.
 * The CharCats tables define the character classes used in the word
 * wrap or click tables.  The BreakTables are finite-state machines
 * that determine word wrapping.  The ClickTables are finite-state
 * machines that determine which characters are selected when the
 * user double clicks.
 */

const unsigned char * const NXEnglishSmartLeftChars;
const unsigned char * const NXEnglishSmartRightChars;
const unsigned char * const NXEnglishCharCatTable;
const NXFSM * const NXEnglishBreakTable;
const int NXEnglishBreakTableSize;
const NXFSM * const NXEnglishNoBreakTable;
const int NXEnglishNoBreakTableSize;
const NXFSM * const NXEnglishClickTable;
const int NXEnglishClickTableSize;

const unsigned char * const NXCSmartLeftChars;
const unsigned char * const NXCSmartRightChars;
const unsigned char * const NXCCharCatTable;
const NXFSM * const NXCBreakTable;
const int NXCBreakTableSize;
const NXFSM * const NXCClickTable;
const int NXCClickTableSize;

typedef int (*NXTextFunc) (id self, NXLayInfo *layInfo);

typedef unsigned short (*NXCharFilterFunc) (unsigned short
    charCode, int flags, unsigned short charSet);

typedef char *(*NXTextFilterFunc) (id self, unsigned char *
    insertText, int *insertLength, int position);
```

TextField

INHERITS FROM	Control : View : Responder : Object
DECLARED IN	appkit/TextField.h

CLASS DESCRIPTION

The TextField class provides a Control object that can display a piece of text, select all or part of it if it is selectable, and edit it if it is editable. It is a good alternative to the Text object when you want small editable text since you don't have to allocate memory for a Text object for each TextField instance—the display of the TextField is achieved by using a global Text object shared by objects all over your application. Moreover, editing and selecting are achieved by a Text object that is unique for a given Window. The TextField is a Control in the sense that the action message of its Cell is sent to the target object of its Cell when the user presses the Return key. When the user presses the Tab key and when there is some object in the TextField's **nextText** instance variable that responds to the **selectText:** method (such as another field of data to enter), that object is selected.

You can drag TextField and an accompanying TextFieldCell into an application from the Interface Builder Palettes panel.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Responder	id	nextResponder;
Inherited from View	NXRect	frame;
	NXRect	bounds;
	id	superview;
	id	subviews;
	id	window;
	struct __vFlags	vFlags;
Inherited from Control	int	tag;
	id	cell;
	struct _conFlags	conFlags;
Declared in TextField	id	nextText;
	id	previousText;
	id	textDelegate;
	SEL	errorAction;

nextText	the object to select when Tab is pressed

previousText	object to select when Shift-Tab is pressed
textDelegate	delegate for **textDidEnd:endChar:**, etc.
errorAction	sent to target when a bad value is entered in the field

METHOD TYPES

Initializing the TextField Class	+ setCellClass:
Initializing a new TextField	– initFrame:
Enabling the TextField	– setEnabled:
Modifying Text Attributes	– isEditable – isSelectable – setEditable: – setSelectable:
Editing Text	– selectText: – setNextText: – setPreviousText: – textDidGetKeys:isEmpty: – textDidChange: – textDidEnd:endChar: – textWillChange: – textWillEnd:
Modifying Graphic Attributes	– backgroundColor – backgroundGray – isBezeled – isBordered – isBackgroundTransparent – setBackgroundColor: – setBackgroundGray: – setBackgroundTransparent: – setBezeled: – setBordered: – setTextColor: – setTextGray: – textColor – textGray
Resizing a TextField	– sizeTo::
Target and Action	– errorAction – setErrorAction:

Handling Events	– acceptsFirstResponder – mouseDown:
Archiving	– read: – write:
Assigning a Delegate	– setTextDelegate: – textDelegate

CLASS METHODS

setCellClass:

+ **setCellClass:***classId*

This method initializes which subclass of TextFieldCell is used in implementing all TextFields. The default is TextFieldCell. If you subclass TextFieldCell to modify the behavior of a TextField, send this message with the class object of your subclass as the argument. Returns the **id** of the TextField class object.

INSTANCE METHODS

acceptsFirstResponder

– (BOOL)**acceptsFirstResponder**

Returns YES if the TextField is editable or selectable, NO otherwise.

See also: **– setEditable:**, **– setSelectable**

backgroundColor

– (NXColor)**backgroundColor**

Returns the background color of the TextField.

backgroundGray

– (float)**backgroundGray**

Returns the background gray.

errorAction

– (SEL)**errorAction**

Returns the action (a selector) that is sent to the target of the TextField upon text-editing errors (for example, if.the user typed something that wasn't acceptable).

See also: – **setErrorAction:**, – **setEntryType:** (Cell)

initFrame:

– initFrame:(const NXRect *)*frameRect*

Initializes and returns the receiver, a new instance of TextField, with default parameters in the given frame. The text is set to "Some Text", the action is set to NULL, and the justification mode is set to NX_LEFTALIGNED. Also by default, the text is editable and the TextField is surrounded by a bezel. This method is the designated initializer for the TextField class.

isBackgroundTransparent

– (BOOL)**isBackgroundTransparent**

Returns YES if the background is transparent.

isBezeled

– (BOOL)**isBezeled**

Returns YES if the text is in a bezeled frame.

isBordered

– (BOOL)**isBordered**

Returns YES if the text has a border around it.

isEditable

– (BOOL)**isEditable**

Returns YES if the text is editable and selectable.

isSelectable

– (BOOL)**isSelectable**

Returns YES if the text is selectable.

mouseDown:

– **mouseDown:**(NXEvent *)*theEvent*

You never invoke this method directly, but may override it to implement subclassses of the TextField class. If the receiver is editable text editing begins; if the receiver is selectable, text is selected as appropriate. Returns **self**.

read:

– **read:**(NXTypedStream *)*stream*

Reads the TextField from the typed stream *stream*. Returns **self**.

selectText:

– **selectText:***sender*

Selects all contents of the receiving TextField if it is editable or selectable. If you invoke this method before inserting the TextField in a view hierarchy, it has no effect. Returns **self**.

setBackgroundColor:

– **setBackgroundColor:**(NXColor)*Colorvalue*

Sets the background color for the TextField. Returns **self**.

setBackgroundGray:

– **setBackgroundGray:**(float)*value*

Sets the background gray for the TextField. Returns **self**.

setBackgroundTransparent:

– **setBackgroundGray:**(BOOL)*flag*

Sets the background of the TextField to transparent. Returns **self**.

setBezeled:

– **setBezeled:**(BOOL)*flag*

If *flag* is YES, then a bezel will be drawn around the text. Returns **self**.

setBordered:

– **setBordered:**(BOOL)*flag*

If *flag* is YES, then a 1-pixel black border will be drawn around the text. Returns **self**.

setEditable:

– **setEditable:**(BOOL)*flag*

If *flag* is YES, then the text in the TextField is made editable and selectable. If NO, then the text cannot be edited; it may, however, be selectable. Returns **self**.

setEnabled:

– **setEnabled:**(BOOL)*flag*

If *flag* is YES, then the TextField is made active; if NO, then the TextField is made inactive. Redraws the text of the cell if autodisplay is on and the enabled state changes. Returns **self**.

setErrorAction:

– **setErrorAction:**(SEL)*aSelector*

Sets the action that is sent to the target of the TextField upon text-editing errors. An error can occur when the user types something into a cell and the value returned when **isEntryAcceptable:** is sent to the cell is NO. This is a convenient method for enforcing some restrictions on what a user can type into a Cell. Returns **self**.

setNextText:

– **setNextText:***anObject*

Sets the **nextText** instance variable to *anObject*. If the *anObject* responds to **setPreviousText:** and **selectText:**, then it is sent a **setPreviousText:** message with **self** as the argument. The **nextText** instance variable is used to determine the TextField's action when the user presses the Tab key; if **nextText** contains an object which responds to **selectText:**, the current TextField is deactivated and the **selectText:** message is sent to *anObject*. Returns **self**.

setPreviousText:

– **setPreviousText:***anObject*

Normally you never use this method directly. It's invoked automatically by some other object's **setNextText:** method. It sets the object that will be sent **selectText:** when Shift-Tab is pressed in the TextField. Returns **self**.

setSelectable:

– **setSelectable:**(BOOL)*flag*

If *flag* is YES, then the TextField is made selectable but not editable. If NO, then the text is made static; neither editable nor selectable. Returns **self**.

See also: **– isEditable**, **– isSelectable**, **– setEditable**

setTextColor:

– **setTextColor:**(NXColor)*Colorvalue*

Sets the color for text in the TextField. Returns **self**.

setTextDelegate:

– **setTextDelegate:***anObject*

Sets the object to which the TextField will forward any messages from the field editor. These messages include **text:isEmpty:**, **textWillEnd:**, **textDidEnd:endChar:**, **textWillChange:**, and **textDidChange:**. Returns **self**.

See also: **– textDelegate**

setTextGray:

– **setTextGray:**(float)*value*

Sets the gray used to draw the text in the TextField. Returns **self**.

sizeTo::

– **sizeTo:**(float)*width* **:**(float)*height*

If editing is occurring in the TextField, this aborts the editing. Then, after the View is resized, this method reselects the text so that editing can continue. Returns **self**.

textColor

– (NXColor)**textColor**

Returns the color of text in the TextField.

textDelegate

– **textDelegate**

Returns the object that receives messages that are forwarded by the TextField from the field editor. This object is set with the **setTextDelegate:** method.

See also: **– setTextDelegate:**

textDidChange:

– **textDidChange:***textObject*

Delegates to the **textDelegate**. Can be overridden. Returns **self**.

textDidEnd:endChar:

– **textDidEnd:***textObject* **endChar:**(unsigned short)*whyEnd*

Invoked automatically when text editing ends. If editing ends because the Return key has been pressed, the TextField's Cell sends its action message to its target. If the Tab key has been pressed, then the **selectText:** method is sent to the object stored in **nextText** or to **self** if **nextText** is **nil**. Returns **self**.

textDidGetKeys:isEmpty:

– **textDidGetKeys:***textObject* **isEmpty:**(BOOL)*flag*

Delegates to the **textDelegate**. You can override this method. Returns **self**.

textGray

– (float)**textGray**

Returns the gray value used to draw the text in the TextField.

textWillChange:

– (BOOL)**textWillChange:***textObject*

Invoked automatically during editing to determine if it is okay to edit this field. This method checks whether the TextField is editable and sends the text delegate a **textWillChange** message to allow it to respond. Returns NO if the text is editable; YES if the text is not editable.

See also: **– setEditable**, **– setTextDelegate**

textWillEnd:

– (BOOL)**textWillEnd:***textObject*

Invoked automatically before text editing ends. This method returns YES if the editing can't end, NO if editing can end. Determines the return value by sending the TextField's cell an **isEntryAcceptable:** message and sending the text delegate a **textWillEnd:** message.

write:

– **write:**(NXTypedStream *)*stream*

Writes the receiving TextField to the typed stream *stream*. Returns **self**.

TextFieldCell

INHERITS FROM	ActionCell : Cell : Object
DECLARED IN	appkit/TextFieldCell.h

CLASS DESCRIPTION

TextFieldCell is used when you want an NX_TEXTCELL that knows what the background and text gray values are. Normally, the Cell class assumes white as the background when bezeled, and light gray otherwise, and black text is always used. With TextFieldCell, you can specify those two parameters. This object is used by TextField.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Cell	char	*contents;
	id	support;
	struct _cFlags1	cFlags1;
	struct _cFlags2	cFlags2;
Inherited from ActionCell	int	tag;
	id	target;
	SEL	action;
Declared in TextFieldCell	float	backgroundGray;
	float	textGray;

backgroundGray	The background gray color
textGray	The gray used to display the text

METHOD TYPES

Initializing a new TextFieldCell	– init
	– initTextCell:
Copying a TextFieldCell	– copy

Modifying Graphic Attributes	– backgroundColor – backgroundGray – isOpaque – setBackgroundColor: – setBackgroundGray: – isBackgroundTransparent: – setBackgroundTransparent: – setBezeled: – setTextAttributes: – setTextColor: – setTextGray: – textColor – textGray
Displaying	– drawInside:inView: – drawSelf:inView:
Tracking the Mouse	– trackMouse:inRect:ofView:
Archiving	– read: – write:

INSTANCE METHODS

backgroundColor

– (NXColor)**backgroundColor**

Returns the color used to draw the background.

backgroundGray

– (float)**backgroundGray**

Returns the gray used to draw the background.

copy

– **copy**

Creates and returns a new TextFieldCell as a copy of the receiver.

drawInside:inView:

– **drawInside:**(const NXRect *)*cellFrame* **inView:***controlView*

Draws the inside of the TextFieldCell only (in other words, it doesn't draw the bezels or border if any). This method is invoked from **drawSelf:inView:** and also from Control and its subclasses' **drawCellInside:** method (which is invoked from Cell's **set***Type***Value:** methods). If you subclass TextFieldCell, and you override **drawSelf:inView:**, then you must override this method as well. Returns **self.**

drawSelf:inView:

– **drawSelf:**(const NXRect *)*cellFrame* **inView:***controlView*

Draws the text with the appropriate **textGray** and **backgroundGray**. Returns **self.**

init

– **init**

Initializes and returns the receiver, a new instance of TextFieldCell, with the default title, "Field". Other defaults are set as described in **initTextCell:** below.

initTextCell:

– **initTextCell:**(const char *)*aString*

Initializes and returns the receiver, a new instance of TextFieldCell, with *aString* as its text. The default **textGray** is NX_BLACK, and the default **backgroundGray** is transparent (–1.0).

This method is the designated initializer for TextFieldCell. Override his method if you create a subclass of TextFieldCell that performs its own initialization. Note that TextFieldCell doesn't override Cell's **initIconCell:** designated initializer; don't use that method to initialize an instance of TextFieldCell.

isBackgroundTransparent:

– (BOOL)**isBackgroundGray:**

Returns YES if the background of the TextFieldCell is transparent.

See also: **– setBackgroundTransparent:**

isOpaque

– (BOOL)**isOpaque**

Returns YES if drawing the cell touches every bit in its frame. This will be true if the cell is bezeled, or if its **backgroundGray** is not transparent.

read:

– **read:**(NXTypedStream *)*stream*

Reads the TextFieldCell from the typed stream *stream*. Returns **self**.

setBackgroundColor:

– **setBackgroundColor:**(NXColor)*Colorvalue*

Sets the background color for the TextFieldCell. Returns s**elf**.

setBackgroundGray:

– **setBackgroundGray:**(float)*value*

Sets the gray that will be used to draw the background. A *value* of less than 0.0 will result in no background being drawn. If the cell is editable, it must have a background gray greater than or equal to 0.0. Returns **self**.

setBackgroundTransparent:

– **setBackgroundGray:**(BOOL)*flag*

Sets the background of the TextFieldCell to transparent. Returns **self**.

setBezeled:

– **setBezeled:**(BOOL)*flag*

Puts a bezel around the text. If the current **backgroundGray** is transparent, it's changed to NX_WHITE. Bezeled transparent TextFields look strange, but if you want to have one, invoke **setBackgroundGray:** with –1.0 AFTER invoking **setBezeled:**.

setTextAttributes:

– **setTextAttributes:***textObj*

You rarely need to override this method; you never need to invoke it. Sets the background and text gray levels. If the cell is disabled, then the gray level is brought toward the background gray by 1/3. For example, if the background gray is white, and the text gray is dark gray, the disabled text gray would be light gray. If the background gray is black and the text gray is white, then the disabled gray would be light gray. Note that if this cell is editable, and you have set the background gray to be transparent (in other words, less than 0.0), then you will get the default background gray (NX_LTGRAY). Also note that a TextFieldCell is transparent by default. Returns *textObj*.

See also: – **setTextGray:**, – **setBackgroundGray:**, – **setTextAttributes:** (Cell)

setTextColor:

– **setTextColor:**(NXColor)*Colorvalue*

Sets the color that will be used to draw the text. Returns **self**.

setTextGray:

– **setTextGray:**(float)*value*

Sets the gray that will be used to draw the text. Returns **self**.

textGray

– (float)**textGray**

Returns the gray that will be used to draw the text. Returns **self**.

trackMouse:inRect:ofView:

– (BOOL)**trackMouse:**(NXEvent*)*event*
        inRect:(const NXRect*)*aRect*
        ofView:*controlView*

Does nothing since clicking in a TextFieldCell causes editing to occur.

write:

– **write:**(NXTypedStream *)*stream*

Writes the receiving TextFieldCell to the typed stream *stream*. Returns **self**.

View

INHERITS FROM	Responder : Object
DECLARED IN	appkit/View.h

CLASS DESCRIPTION

View is an abstract class that provides its subclasses with a structure for drawing and handling events. Most of the classes defined in the Application Kit are direct or indirect subclasses of View.

Every View is assigned to a Window where it can be displayed. All the Views within the Window are arranged in a hierarchy, with each View having a single superview and zero or more subviews. Each View has its own area to draw in and its own coordinate system, expressed as a transformation of its superview's coordinate system. A View can scale, translate, or rotate its coordinates, flip the polarity of its y-axis, or use the same coordinate system as its superview.

A View keeps track of its size and location in two ways: as a frame rectangle (expressed in its superview's coordinate system) and as a bounds rectangle (expressed in its own drawing coordinates). Both are NXRect structures, defined in the header file **appkit/graphics.h**.

INSTANCE VARIABLES

Inherited from Object

```
Class           isa;
```

Inherited from Responder

```
id              nextResponder;
```

Declared in View

```
NXRect          frame;
NXRect          bounds;
id              superview;
id              subviews;
id              window;
struct __vFlags {
    unsigned int    noClip:1;
    unsigned int    translatedDraw:1;
    unsigned int    drawInSuperview:1;
    unsigned int    alreadyFlipped:1;
    unsigned int    needsFlipped:1;
    unsigned int    rotatedFromBase:1;
    unsigned int    rotatedOrScaledFromBase:1;
    unsigned int    opaque:1;
    unsigned int    disableAutodisplay:1;
    unsigned int    needsDisplay:1;
    unsigned int    validGState:1;
    unsigned int    newGState:1;
}               vFlags;
```

frame	A rectangle that specifies the size and location of the View in its superview's coordinate system.
bounds	A rectangle that specifies the size and location of the View in its own coordinate system.
superview	The View's parent in the view hierarchy.
subviews	A List object that lists the View's immediate children in the view hierarchy.
window	The Window where the View is displayed.
vFlags.noClip	YES if drawing is not clipped to the frame.
vFlags.translatedDraw	YES if the bounds rectangle has been translated (that is, the bounds origin is not (0,0)).
vFlags.drawInSuperview	YES if the bounds origin equals the frame origin.
vFlags.alreadyFlipped	YES if the View's superview is flipped.
vFlags.needsFlipped	YES if the View is flipped.
vFlags.rotatedFromBase	YES if the View's coordinates are rotated from base coordinates.
vFlags.rotatedOrScaledFromBase	YES if the View's coordinates are rotated or scaled from base coordinates.
vFlags.opaque	YES if the View is opaque.
vFlags.disableAutodisplay	YES if automatic display is disabled.
vFlags.needsDisplay	YES if the View needs to be displayed.
vFlags.validGState	YES if the View's graphics state is valid.
vFlags.newGState	YES if the View has a new graphics state.

METHOD TYPES

Initializing and freeing View objects	– initFrame: – init – free

Managing the View hierarchy	– addSubview:
	– addSubview::relativeTo:
	– findAncestorSharedWith:
	– isDescendantOf:
	– opaqueAncestor
	– removeFromSuperview
	– replaceSubview:with:
	– subviews
	– superview
	– window
	– windowChanged:
Modifying the frame rectangle	– frameAngle
	– getFrame:
	– moveBy::
	– moveTo::
	– rotateBy:
	– rotateTo:
	– setFrame:
	– sizeBy::
	– sizeTo::
Resizing subviews	– resizeSubviews:
	– setAutoresizeSubviews:
	– setAutosizing:
	– superviewSizeChanged:
Modifying the coordinate system	– boundsAngle
	– drawInSuperview
	– getBounds:
	– isFlipped
	– isRotatedFromBase
	– isRotatedOrScaledFromBase
	– rotate:
	– scale::
	– setDrawOrigin::
	– setDrawRotation:
	– setDrawSize::
	– setFlipped:
	– translate::
Notifying ancestor Views	– descendantFlipped:
	– descendantFrameChanged:
	– notifyAncestorWhenFrameChanged:
	– notifyWhenFlipped:
	– suspendNotifyAncestorWhenFrameChanged:

Converting coordinates	– centerScanRect: – convertPoint:fromView: – convertPoint:toView: – convertPointFromSuperview: – convertPointToSuperview: – convertRect:fromView: – convertRect:toView: – convertRectFromSuperview: – convertRectToSuperview: – convertSize:fromView: – convertSize:toView:
Graphics state objects	– allocateGState – freeGState – gState – initGState – renewGState – notifyToInitGState:
Focusing	– clipToFrame: – doesClip – setClipping: – isFocusView – lockFocus – unlockFocus
Displaying	– canDraw – display – display:: – display::: – displayFromOpaqueAncestor::: – displayIfNeeded – drawSelf:: – getVisibleRect: – isAutodisplay – setAutodisplay: – isOpaque – setOpaque: – needsDisplay – setNeedsDisplay: – shouldDrawColor – update
Scrolling	– adjustScroll: – autoscroll: – calcUpdateRects:::: – invalidate:: – scrollPoint: – scrollRect:by: – scrollRectToVisible:

Managing the cursor	– addCursorRect:cursor: – discardCursorRects – removeCursorRect:cursor: – resetCursorRects
Assigning a tag	– findViewWithTag: – tag
Aiding event handling	– acceptsFirstMouse – hitTest: – mouse:inRect: – performKeyEquivalent:
Icon dragging	– dragFile:fromRect:slideBack:event:
Printing	– printPSCode: – faxPSCode: – copyPSCodeInside:to: – openSpoolFile: – spoolFile:
Setting up pages	– knowsPagesFirst:last: – getRect:forPage: – placePrintRect:offset: – heightAdjustLimit – widthAdjustLimit
Writing conforming PostScript	– beginPSOutput – beginPrologueBBox:creationDate:createdBy: fonts:forWhom:pages:title: – endHeaderComments – endPrologue – beginSetup – endSetup – adjustPageWidthNew:left:right:limit: – adjustPageHeightNew:top:bottom:limit: – beginPage:label:bBox:fonts: – beginPageSetupRect:placement: – drawSheetBorder:: – drawPageBorder:: – addToPageSetup – endPageSetup – endPage – beginTrailer – endTrailer – endPSOutput
Archiving	– awake – read: – write:

INSTANCE METHODS

acceptsFirstMouse

– (BOOL)**acceptsFirstMouse**

Returns whether the View will accept the mouse down event which caused its window to be made the key window. If this method returns YES, all mouse-down events are passed to the View. Otherwise, the View will only receive mouse-down events when its window is the key window. The default behavior is to return NO.

addCursorRect:cursor:

– **addCursorRect:**(const NXRect *)*aRect* **cursor:***anObj*

Adds a cursor rectangle to the View's Window so that the cursor changes when it enters the specified rectangle of the View. You send this message in response to a **resetCursorRects** message. *aRect* describes the cursor rectangle in the View's coordinates. *anObj* is a Cursor object, like NXIBeam or NXArrow. See View's **resetCursorRects** for more information regarding when this message should be sent. Returns **self**.

See also: – **resetCursorRects**

addSubview:

– **addSubview:***aView*

Links *aView* into the View hierarchy by making it a subview of the receiving View, placing it at the end of its subviews list. The receiving View is also made *aView*'s next responder. Returns **nil** if *aView* was not added as a subview because it does not inherit from View. Otherwise, this method returns *aView.*

See also: – **addSubview::relativeTo:**, – **subviews**, – **removeFromSuperview**, – **initFrame:**, – **setNextResponder:** (Responder)

addSubview::relativeTo:

– **addSubview:***aView*
    :(int)*place*
    relativeTo:*otherView*

Links *aView* into the View hierarchy by making it a subview of the receiving View. This method is just like **addSubview:** with the additional flexibility of precise positioning of *aView* within the subview list. *otherView* is a member of the subview list. *place* can be either NX_ABOVE or NX_BELOW, which specifies the placement of *aView* relative to *otherView.* Since subviews are displayed from first to last in the subview list, the last element is "above" all others. If *otherView* is **nil** or is not a member of the subview list, *aView* will be added to the top or bottom of the subview list depending on the value of *place*. This method returns **nil** if *aView* was not added as a subview because it does not inherit from View. Otherwise, it returns *aView.*

See also: – **addSubview:**, – **subviews**, – **removeFromSuperview**, – **initFrame:**, – **setNextResponder:**

addToPageSetup

– **addToPageSetup**

Allows applications to add a scaling operator to the PostScript code generated when printing; if you must add a scaling operator, this is the correct place to do so. This method is invoked by **printPSCode:** and **faxPSCode:**. By default, this method simply returns **self**; this method can be overridden by applications that implement their own pagination.

See also: – **beginPageSetupRect:placement:**

adjustPageHeightNew:top:bottom:limit:

– **adjustPageHeightNew:**(float *)*newBottom*
  top:(float)*oldTop*
  bottom:(float)*oldBottom*
  limit:(float)*bottomLimit*

Adjusts page height for automatic pagination when printing the View. This method is invoked by **printPSCode:** and **faxPSCode:** to set *newBottom*, which will be the new bottom of the strip to be printed for the current page. *oldTop* and *oldBottom* are the current values for the horizontal strip to be printed. *bottomLimit* is the topmost value *newBottom* can be set to. If this limit is exceeded, *newBottom* is set to *oldBottom*. By default this method tries to not let the View be cut in two. All parameters are in the View's own coordinate system. Returns **self**.

adjustPageWidthNew:left:right:limit:

– **adjustPageWidthNew:**(float *)*newRight*
  left:(float)*oldLeft*
  right:(float)*oldRight*
  limit:(float)*rightLimit*

Adjusts page width for automatic pagination when printing the View. This method is invoked by **printPSCode:** and **faxPSCode:** to set *newRight*, which will be the new right edge of the strip to be printed for the current page. *oldLeft* and *oldRight* are the current values for the vertical strip to be printed. *rightLimit* is the leftmost value *newRight* can be set to. If this limit is exceeded, *newRight* is set to *oldRight*. By default this method tries to not let the View be cut in two. All parameters are in the View's own coordinate system. Returns **self**.

adjustScroll:

– **adjustScroll:**(NXRect *)*newVisible*

Allows you to correct the scroll position of a document. This method is invoked by a ClipView immediately prior to scrolling its document view. You may want to override it to provide specific scrolling behavior. *newVisible* will be the visible rectangle after the scroll. You might use this for scrolling through a table as in a spreadsheet. You could modify *newVisible->origin* such that the scroll would fall on column or row boundaries. Returns **self**.

allocateGState

– **allocateGState**

Explicitly tells the View to allocate a graphics state object. Graphics state objects are Display PostScript objects that contain the entire state of the graphics environment. They are used by the Application Kit as a caching mechanism to save PostScript code used for focusing, purely as a performance optimization. You can allocate a graphics state object for Views that will be focused on repeatedly, but you should exercise some discretion as they can take a fair amount of memory. The graphics state object will be freed automatically when the View is freed. Returns **self**.

See also: – **freeGState**

autoscroll:

– **autoscroll:**(NXEvent *)*theEvent*

Scrolls the View when the cursor is dragged to a position outside its superview. You invoke this method from within a modal responder loop to cause scrolling to occur when the cursor is outside the View's superview. The receiving View must be the document view of a ClipView for this method to have any effect. *theEvent->location* must be in window base coordinates. You can invoke this method repeatedly so that scrolling continues even when there is no mouse movement. Returns **nil** if no scrolling occurs; otherwise returns **self**.

See also: – **autoscroll:** (ClipView), – **beginModalSession:for:** (Application)

awake

– **awake**

Invoked after unarchiving to allow the View to perform additional initialization. Returns **self**.

beginPage:label:bBox:fonts:

– **beginPage:**(int)*ordinalNum*
  label:(const char *)*aString*
  bBox:(const NXRect *)*pageRect*
  fonts:(const char *)*fontNames*

Writes a conforming Postscript page separator. This method is invoked by **printPSCode:** and **faxPSCode:**.

ordinalNum specifies 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 NULL, the ASCII 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. If *pageRect* is NULL, "(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 this page. Each name should be separated by a space. If the fonts used are unknown before the page is printed, *fontNames* can be NULL. They will then be listed automatically at the end of the page description. Returns **self**.

beginPageSetupRect:placement:

– **beginPageSetupRect:**(const NXRect *)*aRect*
  placement:(const NXPoint *)*location*

Writes the page setup section for a page. This method is invoked by **printPSCode:** and **faxPSCode:** after the starting comments for the page have been written. It outputs a PostScript **save**, and generates the initial coordinate transformation to set this View up for printing the *aRect* rectangle within the View. This method does a **lockFocus** on the View, which must be balanced in **endPage** by an **unlockFocus**. The **save** output here should be balanced by a PostScript **restore** in **endPage**. *aRect* is the rectangle in the View's coordinates that is being printed. *location* is the offset in page coordinates of the rectangle on the physical page. Returns **self**.

See also: – **printPSCode**, – **endPage**, – **lockFocus**, – **addToPageSetup**

beginPrologueBBox:creationDate:createdBy:fonts:forWhom:pages:title:

– **beginPrologueBBox:**(const NXRect *)*boundingBox*
  creationDate:(const char *)*dateCreated*
  createdBy:(const char *)*anApplication*
  fonts:(const char *)*fontNames*
  forWhom:(const char *)*user*
  pages:(int)*numPages*
  title:(const char *)*aTitle*

Invoked by **printPSCode:** and **faxPSCode:** to write the start of a conforming PostScript header.

boundingBox is the bounding box of the document. This rectangle should be in the default PostScript coordinate system on the page. If it is unknown *boundingBox* should be NULL and the system will accumulate it as pages are printed.

dateCreated is an ASCII string containing a human readable date. If *dateCreated* is NULL the current date is used.

anApplication is a string containing the name of the document creator. If *anApplication* is NULL then the string returned by Application's **appName** method is used.

fontNames is a string holding the names of the fonts used in the document. Names should be separated by a space. If the fonts used are unknown before the document is printed, *fontNames* should be NULL. In this case each font that is referenced by a **findFont** is written in the trailer.

user is a string containing the name of the person the document is being printed for. If NULL the login name of the user is used.

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 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 NULL, then the title of the View's Window is used. If the Window has no title, "Untitled" is output. Returns **self**.

See also: – **appName** (Application)

beginPSOutput

– **beginPSOutput**

Performs various initializations before actual PostScript generation begins. This method makes the Display PostScript context stored in the Application object's global PrintInfo object into the current context. This has the effect of redirecting all PostScript output from the Window Server to the spool file or printer. This method is invoked by **printPSCode:** and **faxPSCode:** just before any PostScript is generated. Returns **self**.

beginSetup

– **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 **printPSCode:** and **faxPSCode:** 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. Returns **self**.

beginTrailer

– **beginTrailer**

Writes the start of a conforming PostScript trailer. This method is invoked by **printPSCode:** and **faxPSCode:** immediately after all pages have been written. Returns **self**.

boundsAngle

– (float)**boundsAngle**

Returns the angle of the View's bounds rectangle relative to its frame rectangle. If the View's coordinate system has been rotated, this angle will be the accumulation of all **rotate:** messages; otherwise, it will be 0.0.

See also: – **rotate:**, – **setDrawRotation:**

calcUpdateRects::::

– (BOOL)**calcUpdateRects:**(NXRect *)*rects*
    :(int *)*rectCount*
    :(NXRect *)*enclRect*
    :(NXRect *)*goodRect*

You invoke this method to generate update rectangles for a subsequent display invocation. *rects* is an array of 3 rectangles, and *rectCount* will be set to the number of rectangles in *rects* that have been filled in, which will be either 0, 1, or 3. *enclRect* is a rectangle that contains the entire area subject to update, and *goodRect* is a rectangle that contains the area that does not need to be updated. *goodRect* will be set to the intersection of *goodRect* and *enclRect*, or to a rectangle with an origin and size of zero if they do not intersect. The update rectangles are computed by finding the area in *enclRect* that isn't included in *goodRect*. After the method invocation, if *rectCount* is 0, no update rectangles were generated. If *rectCount* is 1, the area that needs to be updated is in rects[0]. If *rectCount* is 3, the areas that need to be updated are in *rects*[1] and *rects*[2], and *rects*[0] is the same as *enclRect*.

Returns YES if any update rectangles were generated (in other words, if *rectCount* is greater than zero); otherwise returns NO.

See also: – **scrollRect:by:**, **NXIntersectionRect()**

canDraw

– (BOOL)**canDraw**

Informs you of whether drawing will have any result. You only need to send this message when you want to do drawing, but are not invoking one of the display methods. You should not draw or send the **lockFocus:** message if this returns NO. This method returns YES if your View has a Window object, your View's Window object has a corresponding window on the Window Server, and your Window object is enabled for display; otherwise it returns NO.

See also: – **isDisplayEnabled** (Window)

centerScanRect:

– **centerScanRect:**(NXRect *)*aRect*

Converts the corners of a rectangle to lie on the center of device pixels. This is useful in compensating for PostScript overscanning when the coordinate system has been scaled. This routine 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 View's coordinate system. Returns **self**.

clipToFrame:

– **clipToFrame:**(const NXRect *)*frameRect*

Allows the View to do arbitrary clipping during focusing. This method is invoked from within the focusing mechanism if clipping is required. If you override this method, you must use *frameRect* rather than the View's **frame** instance variable, because the origins may not be the same due to focusing. The following example demonstrates clipping the View to a circular region:

```
- clipToFrame:(const NXRect *)frameRect
{
    float x, y, radius;

    // Center the circle and pick an appropriate radius
    x = frameRect->origin.x + frameRect->size.width/2.0;
    y = frameRect->origin.y + frameRect->size.height/2.0;
    radius = frameRect->size.height/2.0;

    // Create a circular clipping path
    PSnewpath();
    PSarc(x, y, radius, 0.0, 360.0);
    PSclosepath();
    PSclip();

    return self;
}
```

If you override this method, you will probably need to send a **setCopyOnScroll:**NO to the View's subviews to make them scroll properly. Returns **self**.

See also: – **setCopyOnScroll:** (ClipView)

convertPoint:fromView:

– **convertPoint:**(NXPoint *)*aPoint* **fromView:***aView*

Converts a point from *aView*'s coordinate system to the coordinate system of the receiving View. If *aView* == **nil**, then this method converts from window base coordinates. Both *aView* and the receiving View must belong to the same Window. Returns **self**.

convertPoint:toView:

– **convertPoint:**(NXPoint *)*aPoint* **toView:***aView*

Converts a point from the receiving View's coordinate system to the coordinate system of *aView*. If *aView* == **nil**, then this method converts to window base coordinates. Both *aView* and the receiving View must belong to the same Window. Returns **self**.

convertPointFromSuperview:

– **convertPointFromSuperview:**(NXPoint *)*aPoint*

Converts a point from the coordinate system of the receiving View's superview to the coordinate system of the receiving View. Returns **self**.

See also: – **convertRectFromSuperview:**, – **convertPointToSuperview:**

convertPointToSuperview:

– **convertPointToSuperview:**(NXPoint *)*aPoint*

Converts a point from the receiving View's coordinate system to the coordinate system of its superview. Returns **self**.

See also: – **convertPointFromSuperview:**, – **convertPoint:fromView:**

convertRect:fromView:

– **convertRect:**(NXRect *)*aRect* **fromView:***aView*

Converts a rectangle from *aView*'s coordinate system to the coordinate system of the receiving View. *aRect* is a pointer to the rectangle to be converted. Both *aView* and the receiving View must belong to the same Window. Returns **self**.

convertRect:toView:

– **convertRect:**(NXRect *)*aRect* **toView:***aView*

Converts a rectangle from the receiving View's coordinate system to the coordinate system of *aView*. *aRect* is a pointer to the rectangle to be converted. Both *aView* and the receiving View must belong to the same Window. Returns **self**.

convertRectFromSuperview:

– **convertRectFromSuperview:**(NXRect *)*aRect*

Converts a rectangle from the coordinate system of the receiving View's superview to the coordinate system of the receiving View. Returns **self**.

See also: – **convertRectToSuperview:**

convertRectToSuperview:

– **convertRectToSuperview:**(NXRect *)*aRect*

Converts a rectangle from the receiving View's coordinate system to the coordinate system of its superview. Returns **self**.

See also: – **convertRectFromSuperview:**

convertSize:fromView:

– **convertSize:**(NXSize *)*aSize* **fromView:***aView*

Converts *asize* (a vector) from the coordinate system of *aView* to the coordinate system of the receiving View. Both *aView* and the receiving View must belong to the same Window. Returns **self**.

See also: – **convertSize:toView:**

convertSize:toView:

– **convertSize:**(NXSize *)*aSize* **toView:***aView*

Converts *asize* (a vector) from the receiving View's coordinate system to the coordinate system of *aView*. Both *aView* and the receiving View must belong to the same Window. Returns **self**.

See also: – **convertSize:fromView:**

copyPSCodeInside:to:

– **copyPSCodeInside:**(const NXRect *)*rect* **to:**(NXStream *)*stream*

Generates PostScript code for the View and all its subviews for the area indicated by *rect*. The PostScript code is written to the NXStream *stream*. Returns **self**, assuming no exception is raised in the generation of PostScript code. If an exception is raised, control is given to the appropriate error handler, and this method does not return.

See also: **NX_RAISE()**

descendantFlipped:

– **descendantFlipped:***sender*

Notifies the receiving View that *sender*, a View below the receiving View in the view hierarchy, had its coordinate system flipped. A **descendantFlipped:** message is sent from the **setFlipped:** method if a **notifyWhenFlipped:**YES message was previously sent to *sender*.

View's default implementation of this method simply passes the message to the receiving View's superview, and returns the superview's return value. View subclasses should override this method to respond to the message as required. In the Application Kit, ClipView overrides this method to keep its coordinate system aligned with its document view.

See also: – **notifyWhenFlipped:**, – **setFlipped:**, – **descendantFlipped:** (ClipView)

descendantFrameChanged:

– **descendantFrameChanged:***sender*

Notifies the receiving View that *sender*, a View below the receiving View in the view hierarchy, was resized or moved. A **descendantFrameChanged:** message is sent from the **sizeTo::** and **moveTo::** methods if a **notifyAncestorWhenFrameChanged:**YES message was previously sent to *sender*.

View's default implementation of this method simply passes the message to the receiving View's superview, and returns the superview's return value. View subclasses should override this method to respond to the message as required. In the Application Kit, the ClipView class overrides this method to notify the ScrollView to reset scroller knobs when the document view's frame is changed.

See also: – **notifyAncestorWhenFrameChanged:**, – **sizeTo::**, – **moveTo::**

discardCursorRects

– **discardCursorRects**

Removes all cursor rectangles for the View. You rarely invoke this method; typically you invalidate the cursor rectangles which forces them to get reset. Returns **self**.

See also: – **resetCursorRects**, – **discardCursorRects** (Window),
– **invalidateCursorRectsForView:** (Window)

display

– **display**

Displays the View and its subviews. Returns **self**. This method is equivalent to:

```
[<receiver> display:(NXRect *)0 :0 :NO];
```

See also: – **display:::**, – **drawSelf::**

display::

– **display:**(const NXRect *)*rects* **:**(int)*rectCount*

Displays the View and its subviews. The rectangles are specified in the receiving View's coordinate system. Returns **self**. This method is equivalent to:

```
[<receiver> display:rects :rectCount :NO];
```

See also: – **display:::**, – **drawSelf::**

display:::

– **display:**(const NXRect *)*rects*
 :(int)*rectCount*
 :(BOOL)*clipFlag*

Displays the View and its subviews by invoking the **lockFocus**, **drawSelf::**, and **unlockFocus** methods. *rects* is an array of drawing rectangles in the receiving View's coordinate system; they're used to restrict what is displayed. *rectCount* is the number of valid rectangles in *rects* (0, 1, or 3).

If *rectCount* is 3, then *rects*[0] should contain the smallest rectangle that completely encloses *rects*[1] and *rects*[2], the two rectangles that actually specify the regions to be displayed.

If *rectCount* is 1, *rects*[0] should specify the region to be displayed.

If *rectCount* is 0 or *rects* is NULL, the View's visible rectangle is substituted for *rects*[0] and a value of 1 is used for *rectCount*.

In any case, the rectangles in *rects* are intersected against the visible rectangle.

This method doesn't display a subview unless it falls at least partially inside *rects*[0] if *rectCount* is 1, or inside either *rects*[1] or *rects*[2] if *rectCount* is 3. When this method is applied recursively to each subview, the drawing rectangles are translated to the subview's coordinate system and intersected with its bounds rectangle to produce a new array. *rects* and *rectCount* are then passed as arguments to each View's **drawSelf::** method.

If *clipFlag* is YES, this method clips to the drawing rectangles. Clipping isn't done recursively for each subview, however. If this method succeeds in displaying the View, the flag indicating that the View needs to be displayed is cleared. Returns **self**.

See also: – **display**, – **display::**, – **drawSelf::**, – **needsDisplay**, – **update**, – **displayFromOpaqueAncestor:::**

displayFromOpaqueAncestor:::

– **displayFromOpaqueAncestor:**(const NXRect *)*rects*
 :(int)*rectCount*
 :(BOOL)*clipFlag*

Correctly displays Views that aren't opaque. This method searches from the View up the View hierarchy for an opaque ancestor View. The rectangles specified by *rects* are copied and then converted to the opaque View's coordinates and **display:::** is sent to the opaque View. If the receiving View is opaque, this method has the same effect as **display:::**. Returns **self**.

See also: – **display:::**, – **isOpaque**, – **setOpaque:**

displayIfNeeded

– **displayIfNeeded**

Descends the View hierarchy starting at the receiving View and sends a **display** message to each View that needs to be displayed, as indicated by each View's **needsDisplay** flag. This is useful when you wish to disable display in the Window, modify a series of Views, and then display only the ones whose appearance has changed. Returns **self**.

See also: – **display**, – **needsDisplay**

doesClip

– (BOOL)**doesClip**

Returns whether this View will be clipped to its frame when it is drawn. Clipping is on by default.

See also: – **setClipping:**

dragFile:fromRect:slideBack:event:

– **dragFile:**(const char *)*filename*
  fromRect:(NXRect *)*rect*
  slideBack:(BOOL) *aFlag*
  event:(NXEvent *)*event*

Allows a file icon to be dragged from the View to any application that accepts files. You typically invoke this method from within your View's **mouseDown:** method when you receive a mouse event on an icon representing a file. This method sends a message to the WorkSpace Manager, and the WorkSpace Manager takes care of the actual file dragging. The WorkSpace manager finds the icon for *filename* and tracks the mouse. If the file is released over a window that is registered with the WorkSpace Manager, the application for that window will receive an **iconEntered:at...** message. *filename* is the complete name (including path) of the file to be dragged. If there is more than one file to be dragged, you must separate the filenames with a single tab ('\t') character. *rect* describes the position of the icon in the View's coordinates, and the width and height of *rect* must both be 48.0. *aFlag* indicates whether the icon should slide back to its position in the View if the file is not accepted. If *aFlag* is YES and *filename* is not accepted and the user has not disabled icon animation, the icon will slide back; otherwise it will not. *event* describes where the mouse-down event occurred.

This method returns **self** if the View successfully sent the file dragging message to the WorkSpace Manager; otherwise it returns **nil**.

See also: – **mouseDown:** (Responder),
– **iconEntered:at::iconWindow:iconX:iconY:iconWidth:iconHeight:pathList:** (Listener), – **registerWindow:toPort:** (Speaker)

drawInSuperview

– **drawInSuperview**

Makes the View's coordinate system identical to that of its superview. This can reduce the amount of PostScript code that's generated to focus on the View. After invoking this method, the View's bounds rectangle origin is the same as its frame rectangle origin.

Although the View's superview may be flipped, the View's coordinate system won't be flipped unless it receives a **setFlipped:** message. You should invoke **drawInSuperview** after creating the View and before applying any coordinate transformations to it. Returns **self**.

See also: – **setFlipped:**

drawPageBorder::

– **drawPageBorder:**(float)*width* **:**(float)*height*

Allows applications that use the Application Kit pagination facility to draw additional marks on each logical page. This method is invoked by **beginPageSetupRect:placement:**, and the default implementation doesn't draw anything. Returns **self**.

drawSelf::

– **drawSelf:**(const NXRect *)*rects* **:**(int)*rectCount*

Implemented by subclasses to draw the View. Each View subclass must override this method to draw itself within its frame rectangle. The default implementation of this method does nothing.

This method is invoked by the display methods (**display**, **display::**, and **display:::**); you shouldn't send a **drawSelf::** message directly to a View.

rects is an array of rectangles indicating the region within the View that needs to be drawn. *rectCount* indicates the number of rectangles in the *rects* array, which is either 1 or 3. If *rectCount* is 1, then *rects*[0] specifies the region to be drawn. If *rectCount* is 3, then *rects*[0] contains the smallest rectangle that completely encloses *rects*[1] and *rects*[2], the two rectangles that actually specify the regions that need to be drawn. Note that if *rectCount* is 3, you can just draw the contents of *rects*[0], or you can draw the contents of both *rects*[1] and *rects*[2], but there is no need to draw all three rectangles. For optimum drawing performance, you shouldn't draw anything that doesn't intersect with the *rects* rectangles, although it is possible to draw the entire contents of the View and simply allow the contents of the View to be clipped.

Your implementation of **drawSelf::** doesn't need to invoke **lockFocus**; focus is already locked on an object when it's told to draw itself. Returns **self**.

See also: – **display**, – **display::**, – **display:::**

drawSheetBorder::

– **drawSheetBorder:**(float)*width* **:**(float)*height*

Allows applications that use the Application Kit pagination facility to draw additional marks on each printed sheet. This method is invoked by **beginPageSetupRect:placement:**, and the default implementation doesn't draw anything. Returns **self**.

endHeaderComments

– **endHeaderComments**

Writes out the end of a conforming PostScript header. It prints out the %%EndComments line and then the start of the prologue, including the Application Kit's standard printing package. The prologue should contain definitions global to a print job. This method is invoked by **printPSCode:** and **faxPSCode:** after **beginPrologueBBox:creationDate:createdBy:fonts:forWhom:pages:title:** and before **endPrologue**. Returns **self**.

endPage

– **endPage**

Writes the end of a conforming PostScript page. This method is invoked after each page is printed. It performs an **unlockFocus** to balance the **lockFocus** done in **beginPageSetupRect:placement:**. It also generates a PostScript **showpage** and a **restore**. Returns **self**.

See also: – **beginPageSetupRect:placement:**

endPageSetup

– **endPageSetup**

Writes the end of the page setup section, which begins with a %%EndPageSetup comment. This method is invoked by **printPSCode:** and **faxPSCode:** just after **beginPageSetupRect:placement:** is invoked. Returns **self**.

endPrologue

– endPrologue

Writes out the end of the conforming PostScript prologue. This method is invoked by **printPSCode:** and **faxPSCode:** after the prologue of the document has been written. Applications can override this method to add their own definitions to the prologue. For example:

```
- endPrologue
{
    DPSPrintf(DPSGetCurrentContext(), "/littleProc {pop} def");
    return [super endPrologue];
}
```

endPSOutput

– endPSOutput

Ends a print job. This method is invoked by **printPSCode:** and **faxPSCode:**. It closes the spool file (if any), and restores the old PostScript context so that further PostScript output is directed to the Window Server. Returns **self**.

See also: – **beginPSOutput**

endSetup

– endSetup

Writes out the end of the setup section, which begins with a %%EndSetup comment. This method is invoked by **printPSCode:** and **faxPSCode:** just after **beginSetup** is invoked. Returns **self**.

endTrailer

– endTrailer

Writes the end of the conforming PostScript trailer. This method is invoked by **printPSCode:** and **faxPSCode:** just after **beginTrailer** is invoked. Returns **self**.

See also: – **beginTrailer**

faxPSCode:

– **faxPSCode:***sender*

Prints the View and all its subviews to a fax modem. If the user cancels the job, or if there are any errors in generating the PostScript, this method returns **nil**; otherwise it returns **self**.

This method normally brings up the Fax panel before actually initiating printing, but if *sender* implements a **shouldRunPrintPanel:** method, the View will invoke that method to query *sender*. If *sender* then returns NO, then the Fax panel won't be displayed, and the View will be printed using the last settings of the Fax panel.

See also: – **printPSCode:**, – **shouldRunPrintPanel:** (Object methods)

findAncestorSharedWith:

– **findAncestorSharedWith:***aView*

Returns the closest common ancestor in the View hierarchy shared by *aView* and the receiving View, or **nil** if there's no such ancestor. If *aView* and the receiving View are identical, this method returns **self**.

See also: – **isDescendantOf:**

findViewWithTag:

– **findViewWithTag:**(int)*aTag*

Finds a descendant View of the receiving View with a tag of *aTag*. Returns **self** if the receiving View's tag is *aTag*. Otherwise this method recursively looks at the tag of the View's first subview, the first subview's descendants, the View's second subview, and so forth. This method returns the first View with matching tag, or **nil** if no subview or descendant of a subview of the receiving View has a matching tag.

See also: – **tag**

frameAngle

– (float)**frameAngle**

Returns the angle of the View's frame relative to its superview's coordinate system.

See also: – **rotateTo:**, – **rotateBy:**

free

– **free**

Releases the storage for the View and all its subviews. This method also invalidates the cursor rectangles for the View's window, frees the View's graphics state object (if any), and removes the View from the view hierarchy; the View will no longer be registered as a subview of any other View.

See also: – **allocFromZone:** (Object), – **initFrame:**

freeGState

– **freeGState**

Frees the graphics state object that was previously allocated for the View. Returns **self**.

See also: – **allocateGState:**

getBounds:

– **getBounds:**(NXRect *)*theRect*

Copies the View's bounds rectangle into the structure specified by *theRect*. Returns **self**.

See also: – **boundsAngle**

getFrame:

– **getFrame:**(NXRect *)*theRect*

Copies the View's frame rectangle into the structure specified by *theRect*. The frame rectangle is specified in the coordinate system of the View's superview. Returns **self**.

getRect:forPage:

– (BOOL)**getRect:**(NXRect *)*theRect* **forPage:**(int)*page*

Implemented by subclasses to determine the rectangle of the View to be printed for page number *page*. You should override this method to fill in *theRect* with the coordinates of the View (in its own coordinate system) that represent the page requested. The View will later be told to display the *theRect* region in order to generate the image for this page. This method is invoked by **printPSCode:** and **faxPSCode:** if the View's **knowsPagesFirst:last:** method returns YES. The View should not assume that the pages will be generated in any particular order.

This method returns YES if *page* is a valid page number for the View. It returns NO if *page* is outside the View.

See also: – **knowsPagesFirst:last:**

getVisibleRect:

– (BOOL)**getVisibleRect:**(NXRect *)*theRect*

Gets the visible portion of the View. A rectangle enclosing the visible portion is placed in the structure specified by *theRect*. This method returns YES if part of the View is visible, and NO if none of it is.

Visibility is determined by intersecting the View's frame rectangle against the frame rectangles of each of its ancestors in the view hierarchy, after appropriate coordinate transformations. Only those portions of the View that lie within the frame rectangles of all its ancestors can be visible.

If the View is in an off-screen window, or is covered by another window, this method may nevertheless return YES. This method does not take into account any siblings of the receiving View or siblings of its ancestors.

If the View is being printed, this method places the portion of the View that is visible on the page being imaged in the structure specified by *theRect*.

See also: – **isVisible** (Window), – **getDocVisibleRect:** (ScrollView), – **getDocVisibleRect:** (ClipView)

gState

– (int)**gState**

Returns the graphics state object allocated to the View. If no graphics state object has been allocated, or if the View has not been focused on since receiving the **allocateGState** message, this method will return 0. Graphics state objects are not immediately allocated by invoking the **allocateGState** method, but are done in a "lazy" fashion upon subsequent focusing.

See also: – **allocateGState**, – **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 from being cut in half. This limit applies to vertical pagination. This method is invoked by **printPSCode:** and **faxPSCode:**. By default, this method returns 0.2.

See also: – **adjustPageHeightNew:top:bottom:limit:**

hitTest:

– **hitTest:**(NXPoint *)*aPoint*

Returns the subview of the receiving View that contains the point specified by *aPoint*. The lowest subview in the View hierarchy is returned. Returns the View if it contains the point but none of its subviews do, or **nil** if the point isn't located within the receiving View.

This method is used primarily by a Window to determine which View in the View hierarchy should receive a mouse-down event. You'd rarely have reason to invoke this method, but you might want to override it to have a View trap mouse-down events before they get to its subviews.

aPoint is in the receiving View's superview's coordinates.

init

– **init**

Initializes the View, which must be a newly allocated View instance. This method does not alter the default frame rectangle, which is all zeros. This method is equivalent to **initFrame:**NULL. Note that if you instantiate a custom View from Interface Builder, it will be initialized with the **initFrame:** method; initialization code in the **init** method will not be performed. Returns **self**.

See also: – **initFrame:**

initFrame:

– **initFrame:**(const NXRect *)*frameRect*

Initializes the View, which must be a newly allocated View instance. The View's frame rectangle is made equivalent to that pointed to by *frameRect*. This method is the designated initializer for the View class, and can be used to initialize a View allocated from your own zone. Programs generally use instances of View subclasses rather than direct instances of the View class. Returns **self**.

See also: – **init**, + **alloc** (Object), + **allocFromZone:** (Object), + **new** (Object)

initGState

– **initGState**

Implemented by subclasses of View to initialize the View's graphics state. The View will receive this message if you previously sent it a **notifyToInitGState:**YES message. By default this method simply returns **self**, but you can override it to send PostScript code to initialize the View's graphics state. You could use this method to set a default font or line width for the View. You should not use this method to send any coordinate transformations or clipping operators.

See also: – **allocateGState**, – **gState**, – **notifyToInitGState:**

invalidate::

– **invalidate:**(const NXRect *)*rects* **:**(int)*rectCount*

Invalidates the View and its subviews for later display. This message is sent to the View after scrolling if the View is a subview of a ClipView and the View's parent ClipView previously received a **setDisplayOnScroll:**NO message. You can override this method to optimize drawing performance by accumulating the invalid areas for later display. *rects* is an array of rectangles in the receiving View's coordinate system, and *rectCount* is the number of valid rectangles in *rects*.

If *rectCount* is 1, *rects*[0] specifies the region requiring redisplay. If *rectCount* is greater than 1, then *rects*[0] contains the smallest rectangle that completely encloses the remaining rectangles in the *rects* array, which specify the actual regions requiring redisplay. Returns **self**.

See also: – **rawScroll:** (ClipView), – **display**, – **display::**, – **display:::**, – **drawSelf::**, – **setDisplayOnScroll:** (ClipView)

isAutodisplay

– (BOOL)**isAutodisplay**

This method returns the View's automatic display status. After you change your data in such a way that it is no longer accurately represented, you should invoke this method to test the View's automatic display status. If automatic display is enabled, you should send a display message to the View; otherwise you should send it a **setNeedsDisplay:**YES message.

See also: – **update**, – **display**, – **setAutodisplay**, – **needsDisplay**, – **setNeedsDisplay:**, – **displayIfNeeded**

isDescendantOf:

– (BOOL)**isDescendantOf:***aView*

Returns YES if *aView* is an ancestor of the receiving View in the view hierarchy or if it's identical to the receiving View. Otherwise, this method returns NO.

See also: – **superview**, – **subviews**, – **findAncestorSharedWith:**

isFlipped

– (BOOL)**isFlipped**

Returns YES if the receiver uses flipped drawing coordinates or NO if it uses native PostScript coordinates. By default, Views are not flipped.

See also: – **setFlipped:**

isFocusView

– (BOOL)**isFocusView**

Returns YES if the receiving View is the View that's currently focused for drawing; otherwise returns NO. In other words, returns YES if drawing commands will be drawn into this View.

See also: – **lockFocus**

isOpaque

– (BOOL)**isOpaque**

Returns whether the View is opaque. Returns YES if the View guarantees that it will completely cover the area within its frame when it draws itself; otherwise returns NO. This state is useful to ensure correct drawing of invalidated areas.

See also: – **setOpaque:**, – **opaqueAncestor**, – **displayFromOpaqueAncestor:::**

isRotatedFromBase

– (BOOL)**isRotatedFromBase**

Returns YES if the receiving View or any of its ancestors in the View hierarchy have been rotated; otherwise returns NO.

isRotatedOrScaledFromBase

– (BOOL)**isRotatedOrScaledFromBase**

Returns YES if the receiving View or any of its ancestors in the View hierarchy have been rotated or scaled; otherwise returns NO.

knowsPagesFirst:last:

– (BOOL)**knowsPagesFirst:**(int *)*firstPageNum* **last:**(int *)*lastPageNum*

Indicates whether this View can return a rectangle specifying the region that must be displayed to print a specific page. This method is invoked by **printPSCode:** and **faxPSCode:**. Just before invoking this method, the first page to be printed is set to 1, and the last page to be printed is set to the maximum integer size. You can therefore override this method to change the first page to be printed, and also the last page to be printed if the View knows where its pages lie. If this method returns YES, the printing mechanism will later query the View for the rectangle corresponding to a specific page using **getRect:forPage:**.

See also: – **getRect:forPage:**

lockFocus

– (BOOL)**lockFocus**

Locks the PostScript focus on the View so that subsequent graphics commands are applied to the View. This method ensures that the View draws in the correct coordinates and to the correct device. You must send this message to the View before you draw to it, and you must balance it with an **unlockFocus** message to the View when you finish drawing. Returns YES if the focus was already locked on the View, and NO if it wasn't.

lockFocus and **unlockFocus** are sent for you when you display the View with one of the display methods; you don't have to include **lockFocus** or **unlockFocus** in your drawSelf:: method.

See also: – **display:::**, – **isFocusView**, – **unlockFocus**

mouse:inRect:

– (BOOL)**mouse:**(NXPoint *)*aPoint* **inRect:**(NXRect *)*aRect*

Returns whether the cursor hot spot at the point specified by *aPoint* lies inside the rectangle specified by *aRect*. To test if the cursor lies within a specific rectangle, you should use this method rather than using the **NXPointInRect()** function; Cursor events are specified by the coordinates corresponding to the top left corner of the pixel under the cursor, so **NXPointInRect()** may return the wrong result. *aPoint* and *aRect* must be expressed in the same coordinate system.

See also: – **convertPoint:fromView:**, **NXMouseInRect()**, **NXPointInRect()**

moveBy::

– **moveBy:**(NXCoord)*deltaX* **:**(NXCoord)*deltaY*

Moves the origin of the View's frame rectangle by (*deltaX*, *deltaY*) in its superview's coordinates. This method works through the **moveTo::** method. Returns **self**.

See also: – **moveTo::**, – **sizeBy::**

moveTo::

– **moveTo:**(NXCoord)*x* **:**(NXCoord)*y*

Moves the origin of the View's frame rectangle to (*x*, *y*) in its superview's coordinates. This method may also send a **descendantFrameChanged:** message to the View's superview. Returns **self**.

See also: – **setFrame:**, – **sizeTo::**, – **descendantFrameChanged:**

needsDisplay

– (BOOL)**needsDisplay**

Returns whether the View needs to be displayed to reflect changes to its contents. If automatic display is disabled, the View will not redisplay itself automatically, so you can invoke this method to determine whether you need to send a display message to the View. The flag indicating that the View needs to be displayed is cleared by the display methods when the View is displayed.

See also: – **setNeedsDisplay:**, – **update**, – **setAutodisplay**, – **isAutodisplay**, – **display**, – **displayIfNeeded**

notifyAncestorWhenFrameChanged:

– **notifyAncestorWhenFrameChanged:**(BOOL)*flag*

Determines whether the receiving View will inform its ancestors in the view hierarchy whenever its frame changes. If *flag* is YES, subsequent **sizeTo::** and **moveTo::** messages to the View will send a **descendantFrameChanged:** message up the view hierarchy. If *flag* is NO, no **descendantFrameChanged:** message will be sent to the View's ancestors. The **descendantFrameChanged:** message permits Views to make any necessary adjustments when a subview is resized or moved. Returns **self**.

See also: – **descendantFrameChanged:**, – **sizeTo::**, – **moveTo::**

notifyToInitGState:

– **notifyToInitGState:**(BOOL)*flag*

Determines whether the View will be sent **initGState** messages to allow it to initialize new graphics state objects. If *flag* is YES, **initGState** messages will be sent to the View at the appropriate time; otherwise, they will not. By default, the View is not sent messages to initialize its graphics state objects. Returns **self**.

See also: – **initGState**

notifyWhenFlipped:

– **notifyWhenFlipped:**(BOOL)*flag*

Determines whether the receiving View will inform its ancestors in the View hierarchy whenever its coordinate system is flipped. If *flag* is YES, a **setFlipped:** message to the View will send a **descendantFlipped:** message up the View hierarchy. If *flag* is NO, no **descendantFlipped:** message will be sent to the View's ancestors. The **descendantFlipped:** message permits Views to make any necessary adjustments when the orientation of a subview's coordinate system is flipped. Returns **self**.

See also: – **descendantFlipped:**, – **setFlipped:**

opaqueAncestor

– **opaqueAncestor**

Returns the closest ancestor to the receiving View that is an opaque View. This method will return the receiving View if it is opaque.

See also: – **isOpaque**, – **displayFromOpaqueAncestor:::**

openSpoolFile:

– **openSpoolFile:**(char *)*filename*

Opens the *filename* file for print spooling. This method is invoked by **printPSCode:** and **faxPSCode:**; it shouldn't be directly invoked in program code. However, you can override it to modify its behavior.

If *filename* is NULL or an empty string (filename[0] is '\0'), the PostScript code is sent directly to the printing daemon, **npd**, without opening a file. (However, if the Window is being previewed or saved, a default file is opened in /**tmp**).

If *filename* is provided, the file is opened. The printing machinery will then write the PostScript code to that file and the file will be printed (or faxed) using **lpr**.

This method opens a Display PostScript context that will write to the spool file, and sets the context of the application's global PrintInfo object to this new context. It returns **nil** if the file can't be opened; otherwise it returns **self**.

performKeyEquivalent:

– (BOOL)**performKeyEquivalent:**(NXEvent *)*theEvent*

Implemented by subclasses of View to allow them to respond to keyboard input. If the View responds to the key, it should take the appropriate action and return YES. Otherwise, it should return the result of passing the message along to **super**, which will pass the message down the View hierarchy:

```
return [super performKeyEquivalent:theEvent];
```

This method returns YES if the View or any of its subviews responds to the key; otherwise it returns NO.

The default implementation of this method simply passes the message down the View hierarchy and returns NO if none of the View's subviews responds to the key. *theEvent* points to the event record of a key-down event.

See also: – **commandKey:** (Window and Panel)

placePrintRect:offset:

– **placePrintRect:**(const NXRect *)*aRect* **offset:**(NXPoint *)*location*

Determines the location of the rectangle being printed on the physical page. This method is invoked by **printPSCode:** and **faxPSCode:**. *aRect* is the rectangle being printed on the current page. This method sets *location* to be the offset of the rectangle from the lower left corner of the page. All coordinates are in the default PostScript coordinate system of the page.

By default, if the flags for centering are YES in the global PrintInfo object, this routine centers the rectangle within the margins. If the flags are NO, it defaults to abutting the rectangle against the top left margin. Returns **self**.

printPSCode:

– **printPSCode:***sender*

Prints the View and all its subviews. If the user cancels the job, or if there are any errors in generating the PostScript code, this method returns **nil**; otherwise it returns **self**.

This method normally brings up the PrintPanel before actually initiating printing, but if *sender* implements a **shouldRunPrintPanel:** method, the View will invoke that method to query *sender*. If *sender*'s **shouldRunPrintPanel:** method then returns NO, then the PrintPanel will not be brought up as part of the printing process, and the View will be printed using the last settings of the PrintPanel.

See also: – **faxPSCode:**, – **copyPSCodeInside:to:**, – **shouldRunPrintPanel:** (Object methods)

read:

– **read:**(NXTypedStream *)*stream*

Reads the View and its subviews from the typed stream *stream*. Returns **self**.

removeCursorRect:cursor:

– **removeCursorRect:**(const NXRect *)*aRect* **cursor:***anObj*

Removes a cursor rectangle from a window. *aRect* is given in the View's coordinates, and *anObj* is the Cursor object for *aRect*. You rarely need to use this method; it's usually easier to use Window's **invalidateCursorRectsForView:** method and let the **resetCursorRects** mechanism restore the cursor rectangles. Returns **self**.

See also: – **invalidateCursorRectsForView:** (Window), – **resetCursorRects**

removeFromSuperview

– removeFromSuperview

Unlinks the View from its superview and its Window, removes it from the responder chain, and invalidates its cursor rectangles. Returns **self**.

See also: **– addSubview:**

renewGState

– renewGState

Forces the View to reinitialize its graphics state object. This method is lazy; the graphics state object is not refreshed until the View actually draws. Returns **self**.

replaceSubview:with:

– replaceSubview:*oldView* **with:***newView*

Replace *oldView* with *newView* in the View's subview list. This method does nothing and returns **nil** if *oldView* is not a subview of the View or if *newView* is not a View. Otherwise, this method returns *oldView*.

See also: **– addSubview:**

resetCursorRects

– resetCursorRects

Implemented by subclasses to reset the View's cursor rectangles. You never send this message, but this method must be overridden by any View that wants cursor rectangles. When the Application object determines that the key window has invalid cursor rectangles, it sends the **resetCursorRects** message to the key window. The key window then sends the **resetCursorRects** message to each of its subviews. Each View must then send the **addCursorRect:cursor:** message to itself for each visible cursor rectangle. The View must clip the cursor rectangle against the visible rectangle, so your override of this method might look something like this:

```
- resetCursorRects
{
    NXRect visible;
    if ([self getVisibleRect:&visible]) {
        [self addCursorRect:&visible cursor:theCursor];
    }
    return self;
}
```

See also: **– invalidateCursorRectsForView:** (Window), **– getVisibleRect:**, **– addCursorRect:**, **NXIntersectionRect()**

resizeSubviews:

– **resizeSubviews:**(const NXSize *)*oldSize*

Informs the View's subviews that the View's bounds rectangle size has changed. This method is invoked from the **sizeTo::** method if the View has subviews and has received a **setAutoresizeSubviews:**YES message. By default, this method sends a **superviewSizeChanged:** message to each subview. You should not invoke this method directly, but you may want to override it to define a specific retiling behavior. *oldSize* is the previous bounds rectangle size. Returns **self**.

See also: – **sizeTo::**, – **setAutoresizeSubviews:**, – **superviewSizeChanged:**

rotate:

– **rotate:**(NXCoord)*angle*

Rotates the View's drawing coordinates by *angle* degrees from its current angle of orientation. Positive values indicate counterclockwise rotation; negative values indicate clockwise rotation. The position of the coordinate origin, (0.0, 0.0), remains unchanged; it's at the center of the rotation. Returns **self**.

See also: – **translate::**, – **scale::**, – **setDrawRotation:**

rotateBy:

– **rotateBy:**(NXCoord)*deltaAngle*

Rotates the View's frame rectangle by *deltaAngle* degrees from its current angle of orientation. Positive values rotate the frame in a counterclockwise direction; negative values rotate it clockwise. The position of the frame rectangle origin remains unchanged; it's at the center of the rotation. Returns **self**.

See also: – **rotateTo:**

rotateTo:

– **rotateTo:**(NXCoord)*angle*

Rotates the View's frame rectangle to *angle* degrees in its superview's coordinate system. The position of the frame rectangle origin remains unchanged; it's at the center of the rotation. Returns **self**.

See also: – **rotateBy:**

scale::

– **scale:**(NXCoord)*x* **:**(NXCoord)*y*

Scales the View's coordinate system. The length of units along its x and y axes will be equal to *x* and *y* in the View's current coordinate system. Returns **self**.

See also: – **setDrawSize::**, – **translate::**, – **rotate:**

scrollPoint:

– **scrollPoint:**(const NXPoint *)*aPoint*

Scrolls the View, which must be a ClipView's document view. *aPoint* is given in the receiving View's coordinates. After the scroll, *aPoint* will be coincident with the bounds rectangle origin of the ClipView, which is its lower left corner, or its upper left corner if the receiving View is flipped. Returns **self**.

See also: – **setDocView:** (ClipView)

scrollRect:by:

– **scrollRect:**(const NXRect *)*aRect* **by:**(const NXPoint *)*delta*

Scrolls the *aRect* rectangle, which is expressed in the View's drawing coordinates, by *delta*. Only those bits which are visible before and after scrolling are moved. This method works for all Views and does not require that the View's immediate ancestor be a ClipView or ScrollView. Returns **self**.

scrollRectToVisible:

– **scrollRectToVisible:**(const NXRect *)*aRect*

Scrolls *aRect* so that it becomes visible within the View's parent ClipView. The receiving View must be a ClipView's document view. This method will scroll the ClipView the minimum amount necessary to make *aRect* visible. *aRect* is a rectangle in the receiving View's coordinates. Returns **self** if scrolling actually occurs; otherwise returns **nil**.

See also: – **setDocView:** (ClipView)

setAutodisplay:

– **setAutodisplay:**(BOOL)*flag*

Enables or disables automatic display of the View. If *flag* is YES, subsequent messages to the View that would affect its appearance are automatically reflected on the screen. If *flag* is NO, you must explicitly send a display message to reflect changes to the View. By default, changes are automatically displayed. If automatic display is disabled, the

View will set a dirty flag which you can query with the **needsDisplay** method to determine whether you need to send the View a display message. Returns **self**.

See also: – **isAutodisplay**, – **needsDisplay**, – **setNeedsDisplay:**, – **display**, – **update**, – **displayIfNeeded**

setAutoresizeSubviews:

– **setAutoresizeSubviews:**(BOOL)*flag*

Determines whether the **resizeSubviews:** message will be sent to the View upon receipt of a **sizeTo::** message. By default, automatic resizing of subviews is disabled. Returns **self**.

See also: – **resizeSubviews:**, – **sizeTo::**, – **superviewSizeChanged:**

setAutosizing:

– **setAutosizing:**(unsigned int)*mask*

Determines how the receiving View's frame rectangle will change when its superview's size changes. Create *mask* by ORing the following together:

Flag	**Meaning**
NX_NOTSIZABLE	The View does not resize with its superview.
NX_MINXMARGINSIZABLE	The left margin between Views can stretch.
NX_WIDTHSIZABLE	The View's width can stretch.
NX_MAXXMARGINSIZABLE	The right margin between Views can stretch.
NX_MINYMARGINSIZABLE	The top margin between Views can stretch.
NX_HEIGHTSIZABLE	The View's height can stretch.
NX_MAXYMARGINSIZABLE	The bottom margin between Views can stretch.

Returns **self**.

See also: – **sizeTo::**, – **resizeSubviews:**, – **setAutoresizeSubviews:**

setClipping:

– **setClipping:**(BOOL)*flag*

Determines whether drawing is clipped to the View's frame rectangle. Views are clipped by default. When you know the View won't draw outside its frame, you can turn off clipping to reduce the amount of PostScript code sent to the Window Server. You can also use this method to enable clipping in a View that inherits from a subclass that disables clipping. You should send a **setClipping:** message to the View before it first draws, usually from the method that initializes the View. Returns **self**.

See also: – **lockFocus**, – **drawInSuperview**, – **initFrame:**, – **doesClip**

setDrawOrigin::

– **setDrawOrigin:**(NXCoord)*x* **:**(NXCoord)*y*

Translates the View's drawing coordinates so that (*x*, *y*) corresponds to the same point as the View's frame rectangle origin. If the View's drawing coordinates have been rotated or flipped, this won't necessarily coincide with its bounds rectangle origin. Returns **self**.

See also: – **translate::**, – **setDrawSize::**, – **setDrawRotation:**

setDrawRotation:

– **setDrawRotation:**(NXCoord)*angle*

Rotates the View's drawing coordinates around its frame rectangle origin so that *angle* defines the relationship between the View's frame rectangle and its drawing coordinates. Returns **self**.

See also: – **rotate:**, – **setDrawOrigin::**, – **setDrawSize::**

setDrawSize::

– **setDrawSize:**(NXCoord)*width* **:**(NXCoord)*height*

Scales the View's drawing coordinates so that *width* and *height* define the size of the View's frame rectangle in drawing coordinates. If the View's drawing coordinates have been rotated, the View's frame rectangle size won't necessarily be the same as its bounds rectangle size. Returns **self**.

See also: – **scale::**, – **setDrawOrigin::**, – **setDrawRotation:**

setFlipped:

– **setFlipped:**(BOOL)*flag*

Flips the direction of the View's y coordinate. If *flag* is YES, the View's origin will be located at its upper left corner, and coordinate values will increase towards the bottom of the View. You should send a **setFlipped:** message to a View only once, before it draws, usually from the method that initializes it.

Although a View is positioned in its superview's coordinate system, no View will have a flipped coordinate system unless it receives a **setFlipped:**YES message of its own; it can't inherit flipped coordinates from its superview.

This method may also send a **descendantFlipped:** message to the receiving View's superview. Returns **self**.

See also: – **notifyWhenFlipped:**, – **descendantFlipped:**, – **initFrame:**, – **isFlipped**

setFrame:

– **setFrame:**(const NXRect *)*frameRect*

Repositions and resizes the View within its superview's coordinate system by assigning it the frame rectangle specified by *frameRect*. Returns **self**.

See also: – **initFrame:**, – **sizeTo::**, – **moveTo::**

setNeedsDisplay:

– **setNeedsDisplay:**(BOOL)*flag*

This method sets a flag indicating whether the View needs to be displayed. After the View changes its internal state in such a way that it's no longer accurately reflected on the screen, it should query itself with an **isAutodisplay** message. If automatic display is enabled, the View should send a display message to itself. If automatic display is disabled, the View should send a **setNeedsDisplay:**YES message to itself. This message has no effect if automatic display is enabled. Returns **self**.

See also: – **update**, – **setAutodisplay**, – **isAutodisplay**, – **needsDisplay:**, – **display:::**, – **displayIfNeeded**

setOpaque:

– **setOpaque:**(BOOL)*flag*

Registers whether the View is opaque. If the View guarantees it will cover the entire area within its frame when it displays itself, it should send itself a **setOpaque:**YES message. This method is used to ensure correct drawing of invalidated Views. Returns **self**.

See also: – **isOpaque**, – **opaqueAncestor**, – **displayFromOpaqueAncestor:::**

shouldDrawColor

– (BOOL)**shouldDrawColor**

Returns whether the View should be drawn using color. If the View is being drawn to a window and the window can't store color, this method returns NO; otherwise it returns YES.

sizeBy::

– **sizeBy:**(NXCoord)*deltaWidth* **:**(NXCoord)*deltaHeight*

Resizes the View by *deltaWidth* and *deltaHeight* in its superview's coordinates. This method works by invoking the **sizeTo::** method. Returns **self**.

See also: – **sizeTo::**, – **moveBy::**

sizeTo::

– **sizeTo:**(NXCoord)*width* **:**(NXCoord)*height*

Resizes the View's frame rectangle to the specified *width* and *height* in its superview's coordinates. It may also initiate a **descendantFrameChanged:** message to the View's superview. Returns **self**.

See also: – **setFrame:**, – **moveTo::**, – **sizeBy::**, – **descendantFrameChanged:**

spoolFile:

– **spoolFile:**(const char *)*filename*

Spools the generated PostScript file to the printer. This method is invoked by **printPSCode:** and **faxPSCode:**. Returns **self**.

subviews

– **subviews**

Returns the List object that contains the receiving View's subviews. You can use this List to send messages to each View in the View hierarchy. You must not modify this List directly; use **addSubview:** and **removeFromSuperview** to add and remove Views from the View hierarchy.

See also: – **superview**, – **addSubview:**, – **removeFromSuperview**

superview

– **superview**

Returns the receiving View's superview.

See also: – **window**, – **subviews**, – **addSubview:**, – **removeFromSuperview**

superviewSizeChanged:

– **superviewSizeChanged:**(const NXSize *)*oldSize*

Informs the View that its superview's size has changed. This method is invoked when the View's superview has received a **resizeSubviews:** message. This method will automatically resize the View according to the parameters set by the **setAutosizing:** message. You may want to override this method to provide specific resizing behavior. *oldSize* is the previous bounds rectangle size of the receiving View's superview. Returns **self**.

See also: – **resizeSubviews:**, – **sizeTo::**, – **setAutoresizeSubviews:**

suspendNotifyAncestorWhenFrameChanged:

– **suspendNotifyAncestorWhenFrameChanged:**(BOOL)*flag*

Temporarily disables or reenables the sending of **descendantFrameChanged:** messages to the View's superview when the View is sized or moved. You must have previously sent the View a **notifyAncestorWhenFrameChanged:**YES message for this method to have any effect. These messages do not nest. Returns **self.**

See also: – **descendantFrameChanged:**, – **notifyAncestorWhenFrameChanged:**, – **sizeTo::**, – **moveTo::**,

tag

– (int)**tag**

Returns the View's tag, a integer that you can use to identify objects in your application. By default, View returns (-1). You can override this method to identify certain Views. For example, your application could take special action when a View with a given tag receives a mouse event.

See also: – **findViewWithTag:**

translate::

– **translate:**(NXCoord)*x* **:**(NXCoord)*y*

Translates the origin of the View's coordinate system to (*x*, *y*). Returns **self.**

See also: – **setDrawOrigin::**, – **scale::**, – **rotate:**

unlockFocus

– **unlockFocus**

Balances an earlier **lockFocus** message to the same View. If the **lockFocus** method saved the previous graphics state, this method restores it. Returns **self.**

See also: – **lockFocus**, – **display:::**

update

– **update**

Invokes the proper update behavior when the contents of the View have been changed in such a way that they are no longer accurately represented on the screen. If automatic display is enabled, this method invokes **display**; otherwise this method sets a flag indicating that the View needs to be displayed. Returns **self.**

See also: – **setNeedsDisplay**, – **isAutoDisplay**, – **display**, – **displayIfNeeded**

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 from being cut in half. This limit applies to horizontal pagination. This method is invoked by **printPSCode:** and **faxPSCode:**. By default, this method returns 0.2.

See also: – **adjustPageHeightNew:top:bottom:limit:**

window

– **window**

Returns the Window of the receiving View.

See also: – **superview**

windowChanged:

– **windowChanged:***newWindow*

Invoked when the Window the View is in changes (usually from **nil** to non-**nil** or vice versa). This often happens due to a **removeFromSuperview** sent to the View (or some View higher up the hierarchy from it). This method is especially important when the View is the first responder in the window, in which case this method should be overridden to clean up any blinking carets or other first responder dependent activity the View engages in. Note that **resignFirstResponder** is NOT called when a View is removed from the View hierarchy (since the View does not have the opportunity to reject resignation of the first responder). This method is invoked before the **window** instance variable has been changed to *newWindow*. Returns **self**.

write:

– **write:**(NXTypedStream *)*stream*

Writes the receiving View and its subviews to the typed stream *stream*. Returns **self**.

METHODS IMPLEMENTED BY VIEWS THAT ACCEPT COLOR

acceptColor:atPoint:

– **acceptColor:**(NXColor)*color* **atPoint:**(NXPoint *)*aPoint*

Allows a View to accept a color. If your subclass of View implements this method, it will be invoked when the user drags a color (as from an NXColorWell) into your View. Colors are typically dragged using NXColorPanel's **dragColor:withEvent:fromView:** class method. *aPoint* describes the point (in the

View s window s coordinates) to which the color should be applied; you may want to use **convertPoint:fromView:** to convert *aPoint* to the View s coordinates. Your implementation of the **acceptColor:atPoint:** method should take whatever action is appropriate, which may include redisplaying the View.

See also: – **acceptColor:atPoint:** (NXColorWell), – **convertPoint:fromView:**, + **dragColor:withEvent:fromView:** (NXColorPanel), **NXSetColor()**

CONSTANTS AND DEFINED TYPES

```
#define NX_NOTSIZABLE        (0)
#define NX_MINXMARGINSIZABLE (1)
#define NX_WIDTHSIZABLE      (2)
#define NX_MAXXMARGINSIZABLE (4)
#define NX_MINYMARGINSIZABLE (8)
#define NX_HEIGHTSIZABLE     (16)
#define NX_MAXYMARGINSIZABLE (32)

/* Are we drawing, printing, or copying PostScript to the scrap? */

extern short NXDrawingStatus;

/* NXDrawingStatus values */

#define NX_DRAWING  1 /* we re drawing */
#define NX_PRINTING 2 /* we re printing */
#define NX_COPYING  3 /* we re copying to the scrap */
```

Window

INHERITS FROM	Responder : Object
DECLARED IN	appkit/Window.h

CLASS DESCRIPTION

The Window class defines objects that manage and coordinate windows for an application; each object corresponds to a physical window provided by the Window Server. A Window object plays a central role in an application:

- It communicates with the Window Server to create, move, resize, reorder, and free a window on the screen. It also responds to event messages that inform the application that the window has been affected by user actions.
- It manages a hierarchy of Views that draw inside the window and handle all keyboard and mouse events associated with it. It determines how events are assigned to Views and has methods that help regulate the View display mechanism.
- It keeps track of the current status of the window as the key window or main window, as well as its location, size, and other window attributes.
- It provides a text object—a *field editor*—that can be assigned small-scale editing tasks within the window. The field editor is used by NXBrowsers, Forms, Matrices, and TextFields located in the Window.

Rectangles and Views

A Window is defined by a *frame rectangle* that encloses the entire window, including its title bar, resize bar, and border, and by a *content rectangle* that encloses just its content area. Both rectangles are specified in the screen coordinate system.

Corresponding to these rectangles, each Window has at least two Views in its view hierarchy, a *frame view* that fills the entire frame rectangle and draws the border, title bar, and resize bar, and a *content view* that fills the content area. The frame view is the responsibility of the Window object and shouldn't be altered or sent messages by application programs. The content view is the highest accessible View in the Window's view hierarchy; other Views can be installed as its subviews but it can't be made the subview of another View.

Event Handling

The Application object sends mouse and keyboard events to the Window, as well as window-moved, window-exposed, window-resized, and screen-changed subevents of the kit-defined event. The Window object handles the kit-defined subevents itself, and distributes the keyboard and mouse events to View objects in its view hierarchy. A Window receives keyboard events only if it's the key window.

The Window keeps track of the object that was last selected to handle keyboard events as its *first responder*. The first responder is typically the View that displays the current selection. In addition to keyboard events, it's sent action messages that have a user-selected target (a **nil** target in program code). Views that don't display selectable or editable material—such as Buttons, Sliders, and NXSplitViews—and respond only to a limited set of events don't become the first responder. Views that can display a selection—such as a Text object or a Matrix—are potential first responders. The Window continually updates the first responder in response to the user's mouse actions.

Delegates

In addition to its Views and field editor, a Window can have a *delegate* to coordinate activities within the Window and, on occasion, intervene to constrain the Window in some way or respond to action messages the Window receives. The delegate should be provided with methods that can respond to any or all of the notification methods listed under "METHODS IMPLEMENTED BY THE DELEGATE" near the end of this class specification. Before sending a notification message, the Window first checks to see whether the delegate can respond. If not, no message is sent. There's no need to have a delegate implement all the methods.

INSTANCE VARIABLES

```
Inherited from Object      Class                   isa;

Inherited from Responder   id                      nextResponder;

Declared in Window         NXRect                  frame;
                           id                      contentView;
                           id                      delegate;
                           id                      firstResponder;
                           id                      lastLeftHit;
                           id                      lastRightHit;
                           id                      counterpart;
                           id                      fieldEditor;
                           int                     winEventMask;
                           int                     windowNum;
                           float                   backgroundGray;
                           struct _wFlags{
                               unsigned int            style:4;
                               unsigned int            backing:2;
                               unsigned int            buttonMask:3;
                               unsigned int            visible:1;
                               unsigned int            isMainWindow:1;
                               unsigned int            isKeyWindow:1;
                               unsigned int            isPanel:1;
                               unsigned int            hideOnDeactivate:1;
                               unsigned int            dontFreeWhenClosed:1;
                               unsigned int            oneShot:1;
                           }                       wFlags;
                           struct _wFlags2{
                               unsigned int            deferred:1;
                               unsigned int            docEdited:1;
                               unsigned int            dynamicDepthLimit:1;
                           }                       wFlags2;
```

frame	The Window's location and size (its frame rectangle) in screen coordinates.
contentView	The View that fills the content area of the Window.
delegate	The object that receives notification messages from the Window.
firstResponder	The Responder that receives keyboard events and untargeted action messages sent to the Window. The first responder is typically a View in the Window's view hierarchy, the one that displays the current selection, and changes in response to mouse-down events.

lastLeftHit	The last View in the Window's view hierarchy to receive a left mouse-down event.
lastRightHit	The last View in the Window's view hierarchy to receive a right mouse-down event.
counterpart	The **id** of the Window's miniwindow, or, if the Window is a miniwindow, the **id** of the Window it stands for. Since miniwindows aren't created until they're needed, the counterpart may be **nil**. (It will also be **nil** if the Window is a miniworld icon standing for an application.)
fieldEditor	A place holder for a Text object that will display and edit text for any Controls and Cells located within the window.
winEventMask	The events the Window can receive from the Window Server.
windowNum	The window number, used by the Application Kit to identify the window. This number isn't the global number assigned by the Window Server. It corresponds to a user object and is therefore local to the Window's context.
backgroundGray	The background color of the window.
wFlags.style	The style of window; whether it's plain, titled, a miniwindow, or has a frame suitable for a menu.
wFlags.backing	The type of backing for the on-screen display; whether the Window is retained, nonretained, or buffered.
wFlags.buttonMask	Which controls the window has (close button, miniaturize button, or resize bar).
wFlags.visible	True if the window is on-screen (in the screen list).
wFlags.isMainWindow	True when the window is the main window.
wFlags.isKeyWindow	True when the window is the key window.
wFlags.isPanel	True if the window is a panel.
wFlags.hideOnDeactivate	True if the window should be removed from the screen when the application deactivates.

wFlags.dontFreeWhenClosed	True if the Window is not to be freed when closed.
wFlags.oneShot	True if the Window Server should free the window for this object when it's removed from the screen.
wFlags2.deferred	True if the Window Server shouldn't create a window for this object until it's needed.
wFlags2.docEdited	True if the close button indicates that the document has been edited but not saved.
wFlags2.dynamicDepthLimit	True if the window has a dynamic depth limit that can change to match the depth of the display device.

METHOD TYPES

Initializing a new Window instance	– init – initContent:style:backing:buttonMask:defer: – initContent:style:backing:buttonMask: defer:screen:
Freeing a Window object	– free
Setting up the Window	– setTitle: – setTitleAsFilename: – title – setContentView: – contentView – setBackgroundColor: – backgroundColor – setBackgroundGray: – backgroundGray – setHideOnDeactivate: – doesHideOnDeactivate – setMiniwindowIcon: – miniwindowIcon – setOneShot: – isOneShot – setFreeWhenClosed: – setExcludedFromWindowsMenu: – isExcludedFromWindowsMenu

Querying window attributes	– windowNum – buttonMask – style – worksWhenModal – screen – bestScreen
Window status	– makeKeyWindow – makeKeyAndOrderFront: – becomeKeyWindow – isKeyWindow – resignKeyWindow – canBecomeKeyWindow – becomeMainWindow – isMainWindow – resignMainWindow – canBecomeMainWindow
Rectangle support	– getFrame: – getFrame:andScreen: + getFrameRect:forContentRect:style: + getContentRect:forFrameRect:style: + minFrameWidth:forStyle:buttonMask:
Moving and resizing the window	– moveTo:: – moveTo::screen: – moveTopLeftTo:: – moveTopLeftTo::screen: – dragFrom::eventNum: – constrainFrameRect:toScreen: – placeWindow: – placeWindow:screen: – placeWindowAndDisplay: – sizeWindow:: – center
Reordering the window	– makeKeyAndOrderFront: – orderFront: – orderBack: – orderOut: – orderWindow:relativeTo: – isVisible
Converting coordinates	– convertBaseToScreen: – convertScreenToBase:

Managing the display	– display – displayIfNeeded – disableDisplay – isDisplayEnabled – reenableDisplay – flushWindow – flushWindowIfNeeded – disableFlushWindow – reenableFlushWindow – displayBorder – useOptimizedDrawing: – update
Window depths	+ defaultDepthLimit – setDepthLimit: – depthLimit – setDynamicDepthLimit: – hasDynamicDepthLimit – canStoreColor
Graphics state objects	– gState
The field editor	– endEditingFor: – getFieldEditor:for:
Cursor management	– addCursorRect:cursor:forView: – removeCursorRect:cursor:forView: – invalidateCursorRectsForView: – disableCursorRects – enableCursorRects – discardCursorRects – resetCursorRects
Handling user actions and events	– close – performClose: – miniaturize: – performMiniaturize: – deminiaturize: – setDocEdited: – isDocEdited – windowExposed: – windowMoved: – windowResized: – screenChanged:
Setting the event mask	– setEventMask: – addToEventMask: – removeFromEventMask: – eventMask

Aiding event handling	– getMouseLocation: – setTrackingRect:inside:owner:tag:left:right: – discardTrackingRect: – makeFirstResponder: – firstResponder – sendEvent: – rightMouseDown: – commandKey: – tryToPerform:with:
Services menu support	– validRequestorForSendType:andReturnType:
Printing	– printPSCode: – smartPrintPSCode: – faxPSCode: – smartFaxPSCode: – openSpoolFile: – spoolFile: – copyPSCodeInside:to:
Setting up pages	– knowsPagesFirst:last: – getRect:forPage: – placePrintRect:offset: – heightAdjustLimit – widthAdjustLimit
Writing conforming PostScript	– beginPSOutput – beginPrologueBBox:creationDate: createdBy:fonts:forWhom:pages:title: – endHeaderComments – endPrologue – beginSetup – endSetup – beginPage:label:bBox:fonts: – beginPageSetupRect:placement: – endPageSetup – endPage – beginTrailer – endTrailer – endPSOutput
Archiving	– read: – write: – awake
Assigning a delegate	– setDelegate: – delegate

CLASS METHODS

defaultDepthLimit

+ (NXWindowDepth)**defaultDepthLimit**

Returns the default depth limit for the Window. This will be the smaller of:

- The depth of the deepest display device available to the Window Server, or
- The depth set for the application by the NXWindowDepthLimit parameter.

The value returned will be one of these enumerated values (defined in the header file **appkit/graphics.h**):

NX_TwoBitGrayDepth
NX_EightBitGrayDepth
NX_TwelveBitRGBDepth
NX_TwentyFourBitRGBDepth

See also: – **setDepthLimit:**, – **setDynamicDepthLimit:**, – **canStoreColor**

getContentRect:forFrameRect:style:

+ **getContentRect:**(NXRect *)*content*
forFrameRect:(const NXRect *)*frame*
style:(int)*aStyle*

Calculates the content rectangle of a window that occupies, along with its border, title bar, and resize bar, the frame rectangle specified by *frame* and has the style indicated by *aStyle*. The rectangle is returned by reference in the structure specified by *content*. Both rectangles are in screen coordinates. Returns **self**.

Use this method to get a rectangle to pass to the **initContent:style:backing:buttonMask:defer:** method if you want a window (including its border, title bar, and resize bar) to occupy a precise area of the screen. Permitted values for *aStyle* are discussed under that method.

See also: + **getFrameRect:forContentRect:style:**,
– **initContent:style:backing:buttonMask:defer:**

getFrameRect:forContentRect:style:

+ **getFrameRect:**(NXRect *)*frame*
  forContentRect:(const NXRect *)*content*
  style:(int)*aStyle*

Calculates the frame rectangle that will be occupied by a window (including its border, title bar, and resize bar) if it has the content rectangle specified by *content* and the style indicated by *aStyle*. The frame rectangle is returned by reference in the structure specified by *frame*. Both rectangles are in screen coordinates. Returns **self**.

Use this method to be sure the window will fit in the space available to it.

See also: + **getContentRect:forFrameRect:style:**, – **initContent:style:backing:buttonMask:defer:**

minFrameWidth:forStyle:buttonMask:

+ (NXCoord)**minFrameWidth:**(const char *)*aTitle*
  forStyle:(int)*aStyle*
  buttonMask:(int)*aMask*

Returns the minimum width that a Window's frame rectangle must have for it to display all of *aTitle*, given the specified style and button mask. Permitted values for *aStyle* and *aMask* are discussed under **initContent:style:backing:buttonMask:defer:**.

See also: – **initContent:style:backing:buttonMask:defer:**

INSTANCE METHODS

addCursorRect:cursor:forView:

– **addCursorRect:**(const NXRect *)*aRect*
  cursor:*anObject*
  forView:*aView*

Adds the rectangle specified by *aRect* to the Window's list of cursor rectangles, and returns **self**. If the rectangle can't be added (for example, if the rectangle doesn't lie within the content area of the Window), **nil** is returned.

This method is invoked by View's **addCursorRect:cursor:** method, which should be used instead of this method inside of View implementations of the **resetCursorRects** method.

See also: – **addCursorRect:cursor:** (View), – **resetCursorRects** (View)

addToEventMask:

– (int)**addToEventMask:**(int)*newEvents*

Adds *newEvents* to the Window's current event mask and returns the original event mask. (*newEvents* and the original mask are joined through the bitwise OR operator.)

This method is typically used when an object sets up a modal event loop to respond to certain events. The return value should be used to restore the Window's original event mask when the modal loop done.

See also: – **setEventMask:**, – **eventMask**, – **removeFromEventMask:**

awake

– **awake**

Reinitializes the Window object by having the Window Server redisplay the window and assign it an accurate window number. The Window then registers itself in the Application object's window list.

An **awake** message is automatically sent to every object after it has been read in from an archive file and all the objects it refers to are in a usable state. The message gives the object a chance to complete any initialization that **read:** couldn't do. If you override this method in a Window subclass, the subclass method should include a message to incorporate this version of **awake** as well:

```
- awake
{
    [super awake];
    . . .
    return self;
}
```

See also: – **read:**

backgroundColor

– (NXColor)**backgroundColor**

Returns the background color of the window when it's located on a color display device. The default is the color equivalent to the NX_LTGRAY gray value.

See also: – **setBackgroundColor:**

backgroundGray

– (float)**backgroundGray**

Returns the gray displayed in the background of the Window's content area. The default is NX_LTGRAY.

See also: – **setBackgroundGray:**

becomeKeyWindow

– **becomeKeyWindow**

Records the fact that the Window is now the key window, reestablishes its cursor rectangles, and returns **self**. This method passes the **becomeKeyWindow** message on to the Window's first responder, if the first responder implements a method that can respond. The delegate receives a **windowDidBecomeKey:** notification message, if it can respond.

See also: – **resignKeyWindow**, – **becomeMainWindow**, – **setDelegate:**

becomeMainWindow

– **becomeMainWindow**

Records the fact that the receiving Window is now the main window, and returns **self**. This method sends the Window's delegate a **windowDidBecomeMain:** message, if the delegate can respond.

See also: – **resignMainWindow**, – **becomeKeyWindow**, – **setDelegate:**

beginPage:label:bBox:fonts:

– **beginPage:**(int)*ordinalNum*
  label:(const char *)*aString*
  bBox:(const NXRect *)*pageRect*
  fonts:(const char *)*fontNames*

Writes a conforming PostScript page separator. This method is invoked automatically when printing (or faxing) the Window; it should not be used in program code. However, you can override it to modify the separator that it writes.

ordinalNum specifies the position of the page in the document (from 1 through n for an n-page document).

aString is a string that identifies the page according to the document's internal numbering scheme. It should contain no white space characters. If *aString* is NULL, the ASCII equivalent of *ordinalNum* is used.

pageRect is a pointer to the rectangle, in the default user coordinate system, enclosing all marks on the page about to be printed. If *pageRect* is NULL, bounding box information for the page isn't written. Instead, the string "(atend)" is written to indicate that the **endPage** method will write the bounding box at the end of the page description.

fontNames is a string listing the names of the fonts used on the page. The names should be separated by spaces. If the fonts used are unknown before the page is printed, *fontNames* will be NULL. The **endPage** method will then list the fonts at the end of the page description.

See also: – **endPage**, – **printPSCode:**

beginPageSetupRect:placement:

– **beginPageSetupRect:**(const NXRect *)*aRect*
  placement:(const NXPoint *)*location*

Writes the page setup section for a given page. This method is invoked when printing (or faxing) the Window after the starting comments for the page have been written; it should not be used in program code. However, you can override it to modify the section that it writes.

This method writes out the PostScript **save** operator and generates the initial coordinate transformation to prepare for printing the *aRect* rectangle within the Window. The **save** operation is balanced by a **restore** that the **endPage** method writes. The *aRect* rectangle is in the Window's base coordinate system. *location* is the offset of the rectangle from the lower left corner of the physical page; it's specified in page coordinates (equal to units of the base coordinate system).

See also: – **endPageSetup**, – **endPage**, – **printPSCode:**

beginPrologueBBox:creationDate:createdBy:fonts:forWhom:pages:title:

– **beginPrologueBBox:**(const NXRect *)*boundingBox*
  creationDate:(const char *)*dateCreated*
  createdBy:(const char *)*anApplication*
  fonts:(const char *)*fontNames*
  forWhom:(const char *)*user*
  pages:(int)*numPages*
  title:(const char *)*aTitle*

Writes the start of a conforming PostScript header. This method is invoked when printing (or faxing) the Window; it should not be used in program code. However, you can override it to modify the header it writes.

boundingBox is a pointer to the bounding box of the document. This rectangle should be in the default user coordinate system (identical to the Window's base coordinate system but with the origin at the lower left corner of the page). If the bounding box is unknown, *boundingBox* will be NULL. The system will then accumulate it as pages are printed.

dateCreated is an ASCII string containing a human-readable date. If it's NULL, the current date is used.

anApplication is a string containing the name of the document creator. If it's NULL, the string returned by the Application object's **appName** method is used.

fontNames is a string holding the names of the fonts used in the document. Names should be separated by a space. If the fonts used are unknown before the document is printed, *fontNames* will be NULL. In this case, each font that there's a **findfont** operation for will be written in the trailer.

user is a string containing the name of the person printing the document. If it's NULL, the login name of the user is used.

numPages specifies the number of pages in the document. If unknown at the beginning of printing, it has a value of –1. In this case, the pages are counted as they're generated and the total is written in the trailer.

aTitle is a string specifying the title of the document. If *aTitle* is NULL, the Window's title is used.

See also: – **endPrologue**, – **endHeaderComments**, – **printPSCode:**

beginPSOutput

– **beginPSOutput**

Performs various initializations to prepare for generating PostScript code. This method is invoked when printing (or faxing) the Window; it should not be used in program code. However, you can override it to modify or add to the initialization it does.

This method first makes the Display PostScript context stored in the global PrintInfo object (the one returned by NXApp's **printInfo** method) the current context. This has the effect of redirecting all PostScript output from the Window Server to the spool file or printer.

See also: – **endPSOutput**, – **printPSCode:**

beginSetup

– **beginSetup**

Writes the beginning of the document setup section. This method is invoked when printing (or faxing) the Window; it should not be used in program code. However, you can override it to modify the way it writes the section.

The document setup section is intended for general initialization code and to set up the output device. It follows the document prologue but precedes any pages that are to be printed. At the beginning of the section, this method writes a "%%BeginSetup" comment and a "%%PaperSize" comment declaring the type of paper being used. It also writes comments after querying the PrintInfo object with **isManualFeed** and **resolution** messages.

See also: – **endSetup**, – **printPSCode:**

beginTrailer

– **beginTrailer**

Writes the start of a conforming PostScript trailer, and returns **self**. This method is invoked when printing (or faxing) the Window after all the pages have been written; it should not be used in program code. However, you can override it to modify the trailer it writes.

See also: – **endTrailer**, – **printPSCode:**

bestScreen

– (const NXScreen *)**bestScreen**

Returns a pointer to the deepest screen that the Window currently is on, or NULL if the Window is currently off-screen. A Window can be on more than one screen if the user drags it so that it's displayed partly on one device and partly on another.

See also: – **screen**, – **colorScreen** (Application)

buttonMask

– (int)**buttonMask**

Returns a mask that indicates which buttons appear in the Window's title bar and whether the Window has a resize bar. You can test the return value against these constants:

NX_CLOSEBUTTONMASK
NX_RESIZEBUTTONMASK
NX_MINIATURIZEBUTTONMASK

See also: – **initContent:style:backing:buttonMask:defer:**

canBecomeKeyWindow

– (BOOL)**canBecomeKeyWindow**

Returns YES if the receiving Window can be made the key window, and NO if it can't.

See also: – **isKeyWindow**

canBecomeMainWindow

– (BOOL)**canBecomeMainWindow**

Returns YES if the receiving Window can be made the main window, and NO if it can't. A Window can become the main window if it's in the screen list, isn't a Panel, and accepts keyboard events.

See also: – **isMainWindow**

canStoreColor

– (BOOL)**canStoreColor**

Returns YES if the Window has a depth limit that would allow it to store color values, and NO if it doesn't.

See also: – **depthLimit**, – **shouldDrawColor** (View)

center

– **center**

Moves the window to the center of the screen. This is used when putting up modal panels by Application's **runModalFor:** method. Returns **self**.

close

– **close**

Removes the Window from the screen. If the Window is to be freed when it's closed (the default), this method goes on to remove the Window object from the Application object's list of Windows, have the Window Server destroy the window, and send the object a **free** message.

This method is invoked by the Application Kit when the user clicks the Window's close button. You should invoke it only when you have no other use for the Window (unless the Window is not to be freed when it's closed).

Returns **nil**.

See also: – **close** (Menu), – **setFreeWhenClosed:**

commandKey:

– (BOOL)**commandKey:**(NXEvent *)*theEvent*

Returns NO, to indicate that no objects within the Window can handle Command key-down events.

If a Window has any Views that might want to respond to the key-down event as a keyboard alternative, it must override this version of the method and initiate a **performKeyEquivalent:** message to the Views. For example:

```
- (BOOL)commandKey:(NXEvent *)theEvent
{
    if ( [contentView performKeyEquivalent:theEvent] )
        return( YES );
    else
        return( NO );
}
```

The Panel class implements a method like this so that the controls within a panel and the commands within a menu can respond to keyboard alternatives.

A **commandKey:** message is initiated by the Application object when it receives a key-down event while the Command key is pressed. It sends the message to each Window in its window list, until one of them responds YES. A Window doesn't have to be on-screen to receive the message.

The argument, *theEvent*, is a pointer to the key-down event.

See also: – **performKeyEquivalent:** (View), – **commandKey:** (Panel)

constrainFrameRect:toScreen:

– (BOOL)**constrainFrameRect:**(NXRect *)*theFrame*
  toScreen:(NXScreen *)*screen*

Modifies the frame rectangle of the Window so that enough of it will appear on the specified screen to give users control over the Window's title bar. If *screen* is NULL, the Window is constrained to the nearest screen.

A **constrainFrameRect:toScreen:** message is sent to a titled Window (with or without a resize bar) whenever it's placed on-screen or resized by the application. The proposed frame rectangle for the Window is passed in the structure referred to by *theRect*. If this method modifies the rectangle, it returns YES. Otherwise, it returns NO.

You can override this method to prevent a particular Window from being constrained to the screen, or to constrain it differently.

contentView

– **contentView**

Returns the **id** of the Window's current content view.

See also: – **setContentView:**

convertBaseToScreen:

– **convertBaseToScreen:**(NXPoint *)*aPoint*

Converts the point referred to by *aPoint* from the Window's base coordinate system to the screen coordinate system, and returns **self**.

See also: – **convertScreenToBase:**

convertScreenToBase:

– **convertScreenToBase:**(NXPoint *)*aPoint*

Converts the point referred to by *aPoint* from the screen coordinate system to the Window's base coordinate system, and returns **self**.

See also: – **convertBaseToScreen:**

copyPSCodeInside:to:

– **copyPSCodeInside:**(const NXRect *)*rect* **to:**(NXStream *)*stream*

Generates PostScript code for all the Views located inside the *rect* portion of the Window. The rectangle is specified in the Window's base coordinates. The PostScript code is written to *stream.*

This method generates PostScript code in the same way that **printPSCode:** and **faxPSCode:** do, except that it writes it to *stream.* If an exception is raised, it doesn't return.

See also: – **printPSCode:**, – **faxPSCode:**

delegate

– **delegate**

Returns the Window's delegate, or **nil** if it doesn't have one.

See also: – **setDelegate:**

deminiaturize:

– **deminiaturize:***sender*

Removes the receiving miniwindow from the screen and places the real Window at the front of its tier. The value passed in *sender* is ignored. Returns **self**.

See also: – **miniaturize:**

depthLimit

– (NXWindowDepth)**depthLimit**

Returns the depth limit of the Window. This will be one of the following enumerated values (defined in the header file **appkit/graphics.h**):

NX_DefaultDepth
NX_TwoBitGrayDepth
NX_EightBitGrayDepth
NX_TwelveBitRGBDepth
NX_TwentyFourBitRGBDepth

If the return value is NX_DefaultDepth, you can find out what depth that corresponds to by sending the Window class a **defaultDepthLimit** message.

See also: + **defaultDepthLimit**, – **setDepthLimit:**, – **setDynamicDepthLimit:**

disableCursorRects

– **disableCursorRects**

Disables all cursor rectangle management within the Window. Typically this method is used when you need to do some special cursor manipulation, and you don't want the Application Kit interfering. Returns **self**.

See also: – **enableCursorRects**

disableDisplay

– disableDisplay

Prevents the display methods defined in the View class from displaying any Views within the Window. This permits you to alter or update the Views before displaying them again.

Displaying should be disabled only temporarily. Each **disableDisplay** message should be paired with a subsequent **reenableDisplay** message. Pairs of these messages can be nested; drawing won't be reenabled until the last (unnested) **reenableDisplay** message is sent.

Returns **self**.

See also: **– reenableDisplay**, **– isDisplayEnabled**, **– display:::** (View)

disableFlushWindow

– disableFlushWindow

Disables the **flushWindow** method for the Window. If the Window is a buffered window, drawing won't automatically be flushed to the screen by the display methods defined in the View class. This permits several Views to be displayed before the results are shown to the user.

Flushing should be disabled only temporarily, while the Window's display is being updated. Each **disableFlushWindow** message should be paired with a subsequent **reenableFlushWindow** message. Message pairs can be nested; flushing won't be reenabled until the last (unnested) **reenableFlushWindow** message is sent.

Returns **self**.

See also: **– reenableFlushWindow**, **– flushWindow**, **– disableDisplay**

discardCursorRects

– discardCursorRects

Removes all cursor rectangles from the Window, and returns **self**. This method is invoked by **resetCursorRects** to clear out existing cursor rectangles before resetting them. In general, you wouldn't invoke it in the code you write, but might want to override it to change its behavior.

See also: **– resetCursorRects**

discardTrackingRect:

– **discardTrackingRect:**(int)*trackNum*

Removes the tracking rectangle identified by the *trackNum* tag through a call to **PScleartrackingrect()**, and returns **self**. The tag was assigned when the tracking rectangle was created.

See also: – **setTrackingRect:inside:owner:tag:left:right:**

display

– **display**

Displays all drawing done within the window, including the border, resize bar, and title bar. Each visible View within the Window's view hierarchy will receive a display message. If displaying had been disabled within the Window, this method reenables it. Returns **self**.

See also: – **display** (View), – **disableDisplay**, – **displayIfNeeded**

displayBorder

– **displayBorder**

Redraws the Window's border, title bar, and resize bar, and returns **self**. This is normally done automatically for you.

See also: – **display**

displayIfNeeded

– **displayIfNeeded**

Descends the view hierarchy in the Window, sending a **display** message to each View that has been tagged as needing to be updated (that has its **needsDisplay** flag set). This method is useful when you want to disable displaying in the Window, modify a series of Views, then display only the ones that were modified. Returns **self**.

See also: – **display**, – **setNeedsDisplay:** (View), – **update** (View)

doesHideOnDeactivate

– (BOOL)**doesHideOnDeactivate**

Returns YES if the Window will disappear from the screen when the application is deactivated, and NO if it won't.

See also: – **setHideOnDeactivate:**

dragFrom::eventNum:

– **dragFrom:**(float)*x*
  :(float)*y*
  eventNum:(int)*num*

Lets the user drag a window from a point within its interior. By default, users can drag any window that has a title bar. If you want the user to be able to drag a window without a title bar, you can design a View that will invoke this method when it receives a mouse-down event. The Window Server will intercept subsequent mouse-dragged events, move the window to its new position, and inform the application through a window-moved subevent when the user releases the mouse button.

The first two arguments, (*x*, *y*), give the cursor's location in base coordinates. The third argument, *num*, is the event number for the mouse-down event. All three can be taken directly from the event record for the mouse-down event. Returns **self**.

See also: – **moveTo::**

enableCursorRects

– **enableCursorRects**

Reenables cursor rectangle management that had been disabled by the **disableCursorRects** method. Returns **self**.

See also: – **disableCursorRects**

endEditingFor:

– **endEditingFor:***anObject*

Makes the Window's field editor (a Text object) available for a new editing assignment by detaching it from the object it's currently serving (normally its superview and delegate). If the field editor is the first responder, the Window is made the new first responder. This forces a **textDidEnd:endChar:** message to be sent to the field editor's delegate. The field editor then is assigned a **nil** delegate and is removed from the view hierarchy (its superview is made **nil**). This forces an end to editing even if the field editor had refused to resign its status as the first responder.

To conditionally end editing, first try to make the Window the first responder:

```
if ( [myWindow makeFirstResponder:myWindow ] ) {
    [myWindow endEditingFor:nil];
    . . .
}
```

makeFirstResponder: returns **nil** if the current first responder won't resign. This is the preferred way to verify all fields when an OK button is pressed in a panel, for example.

Returns **self**.

See also: – **getFieldEditor:for:**

endHeaderComments

– **endHeaderComments**

Writes out the end of a conforming PostScript header. This method is invoked when printing (or faxing) the Window; it should not be invoked in program code. However, you can override it to modify the comments it writes or add to the beginning of the document prologue. The prologue contains definitions global to a print job.

This method writes the "%%EndComments" line and then writes the Application Kit's standard printing package to begin the prologue proper. If there's an error in writing the package, an NX_printPackageError exception is raised and this method will not return.

See also: – **printPSCode:**,
– **beginPrologueBBox:creationDate:createdBy:fonts:forWhom:pages:title:**

endPage

– **endPage**

Writes the end of a conforming PostScript page. This method is invoked after each page is written when printing (or faxing) the Window; it should not be used in program code. However, you can override it to modify what it writes.

This method generates a **restore** operation after each page has been described and a **showpage** operation when there are no more pages to be printed on the current sheet of paper.

See also: – **beginPage:label:bBox:fonts:**, – **beginPageSetupRect:placement:**,
– **printPSCode:**

endPageSetup

– **endPageSetup**

Writes the "%%EndPageSetup" comment to end the page setup section. This method is invoked automatically when printing (or faxing) the Window; it should not be used in program code. However, you can override it to modify or add to what it writes.

See also: – **beginPageSetupRect:placement:**, – **printPSCode:**

endPrologue

– **endPrologue**

Writes the end of a conforming PostScript prologue. This method is invoked when printing (or faxing) the Window; it should not be used in program code. However, you can override it to modify the end of the prologue.

See also: – **printPSCode:**,
– **beginPrologueBBox:creationDate:createdBy:fonts:forWhom:pages:title:**

endPSOutput

– **endPSOutput**

Finishes a print job by closing the spool file (if any) and restoring the display context so that further PostScript code will be directed to the Window Server. This method is invoked when printing (or faxing) the Window; it should not be used in program code. However, you can override it to modify its behavior.

See also: – **beginPSOutput**, – **printPSCode:**

endSetup

– **endSetup**

Writes the "%%EndSetup" comment that terminates the document setup section. This method is invoked when printing (or faxing) the Window; it should not be used in program code. However, you can override it to add to what it writes.

See also: – **beginSetup**, – **printPSCode:**

endTrailer

– **endTrailer**

Writes a PostScript conforming trailer. This method is invoked when printing (or faxing) the Window; it should not be used in program code. However, you can override it to modify or add to the trailer it writes.

See also: – **beginTrailer**, – **printPSCode:**

eventMask

– (int)**eventMask**

Returns the current event mask for the Window. Use this method when you need to know which types of events the Window Server might associate with the window and send to the application.

See also: – **setEventMask:**, – **addToEventMask:**, – **removeFromEventMask:**

faxPSCode:

– **faxPSCode:***sender*

Prints the Window (all the Views in its view hierarchy including the frame view) to a fax modem. A return value of **nil** indicates that there were errors in generating the PostScript code or that the user canceled the job.

In the current user interface, faxing is initiated from within the Print panel. However, with this method, you can provide users with an independent control for faxing a Window.

This method normally brings up the Fax panel before actually beginning printing. But if *sender* implements a **shouldRunPrintPanel:** method, that method will be invoked to first query whether to run the panel. If **shouldRunPrintPanel:** returns NO, the Fax panel won't be displayed, and the Window will be printed using the last settings of the panel.

See also: – **smartFaxPSCode:**, – **printPSCode:**, – **shouldRunPrintPanel:** (Object Methods)

firstResponder

– **firstResponder**

Returns the current first responder for the Window.

See also: – **makeFirstResponder:**, – **acceptsFirstResponder** (Responder)

flushWindow

– **flushWindow**

Flushes the Window's off-screen buffer to the screen, if the receiving Window is a buffered window and flushing hasn't been disabled by **disableFlushWindow**. This message is automatically invoked when you send the **display** message to a View. Returns **self**.

See also: – **display::** (View), – **disableFlushWindow**

flushWindowIfNeeded

– **flushWindowIfNeeded**

Flushes the Window's off-screen buffer to the screen if the receiving Window is a buffered window, flushing isn't temporarily disabled, and there were some previous **flushWindow** messages that had no effect because flushing was disabled. Using this method after a **reenableFlushWindow** message, rather than using **flushWindow**, will help eliminate unnecessary calls to the Window Server. Returns **self**.

See also: – **flushWindow**, – **disableFlushWindow**, – **reenableFlushWindow**

free

– **free**

Deallocates memory for the Window object, for all the objects in its view hierarchy, and for all its instance variables, including the field editor.

getFieldEditor:for:

– **getFieldEditor:**(BOOL)*flag* **for:***anObject*

Returns the field editor, the Text object associated with the Window. If there's no field editor and *flag* is YES, this method creates a new Text object and assigns it to the **fieldEditor** instance variable before returning the new object's **id**. If *flag* is NO, the current value of the **fieldEditor** instance variable is returned, even if **nil**.

The **fieldEditor** remains **nil** until a Text object is created with this method.

Before returning the field editor, this method sends the Window's delegate a **windowWillReturnFieldEditor:toObject:** message, giving it a chance to substitute another object for the field editor. If it does, the substitute will be returned instead of the field editor. The substitute is not assigned to the **fieldEditor** instance variable.

By making the field editor a temporary subview and becoming its temporary delegate, Controls such as a TextField are able to use its services for entering, editing, and selecting text. Other Views can use it in the same way.

See also: – **endEditingFor:**

getFrame:

– **getFrame:**(NXRect *)*theRect*

Places the Window's frame rectangle—its location and size in screen coordinates—in the rectangle specified by *theRect*, and returns **self**.

See also: – **getFrame:andScreen:**

getFrame:andScreen:

– **getFrame:**(NXRect *)*theRect* **andScreen:**(const NXScreen *)*theScreen*

Copies the Window's frame rectangle into the structure referred to by *theRect*. The screen where the Window is located is provided in the structure referred to by *theScreen*. The frame rectangle is specified relative to the lower left corner of the screen. However, if *theScreen* is NULL, the frame rectangle is specified in absolute coordinates (relative to the origin of the screen coordinate system). Returns **self**.

See also: – **getFrame:**

getMouseLocation:

– **getMouseLocation:**(NXPoint *)*thePoint*

Places the current location of the cursor in the structure specified by *thePoint*. Usually, this information is available somewhere else, such as in the current event record. But when the event record isn't recent enough or is unavailable, you can use this method to get the location from the Window Server. The location is provided in the Window's base coordinate system. Returns **self**.

See also: – **currentEvent** (Application)

getRect:forPage:

– (BOOL)**getRect:**(NXRect *)*theRect* **forPage:**(int)*page*

Implemented by subclasses to provide the rectangle to be printed for page number *page*. A Window receives **getRect:forPage:** messages when it's being printed (or faxed) if its **knowsPagesFirst:last:** method returns YES.

If *page* is a valid page number for the Window, this method should return YES after providing (in the variable referred to by *theRect*) the rectangle that represents the page requested. The rectangle should be specified in the Window's base coordinates.

If *page* is not a valid page number, this method should return NO. By default, it returns NO.

The Window may receive a series of **getRect:forPage:** messages, one for each page that's being printed. It should not assume that the pages will be generated in any particular order.

See also: – **knowsPagesFirst:last:**, – **printPSCode:**

gState

– (int)**gState**

Returns the PostScript graphics state object associated with the Window.

hasDynamicDepthLimit

– (BOOL)**hasDynamicDepthLimit**

Returns YES if the Window's depth limit can change when it changes screens, and NO if it can't.

See also: – **setDynamicDepthLimit:**

heightAdjustLimit

– (float)**heightAdjustLimit**

Returns the fraction of a page that can be pushed onto the next page to prevent items from being cut in half. The limit applies to vertical pagination. By default, it's 0.2.

This method is invoked during automatic pagination when printing (or faxing) the Window; it should not be used in program code. However, you can override it to return a different value. The value returned should lie between 0.0 and 1.0 inclusive.

See also: – **widthAdjustLimit**

init

– **init**

Initializes the receiver, a newly allocated Window object, by passing default parameters to the **initContent:style:backing:buttonMask:defer:** method. The initialized object is a plain, buffered window, and has a default frame rectangle. Returns **self**.

See also: – **initContent:style:backing:buttonMask:defer:**

initContent:style:backing:buttonMask:defer:

– **initContent:**(const NXRect *)*contentRect*
  style:(int)*aStyle*
  backing:(int)*bufferingType*
  buttonMask:(int)*mask*
  defer:(BOOL)*flag*

Initializes the Window object immediately after it has been allocated by Object's **alloc** or **allocFromZone:** method, and returns **self**. This method is the designated initializer for the Window class. Its five arguments specify the Window's frame rectangle, style, buffering type, controls, and whether or not the Window Server will defer creating a window for the object until it's needed.

The first argument, *contentRect*, specifies the location and size of the Window's content area in screen coordinates. If a NULL pointer is passed for this argument, a default rectangle is used.

The second argument, *aStyle*, specifies the window's style. It can be:

NX_PLAINSTYLE
NX_TITLEDSTYLE
NX_RESIZEBARSTYLE
NX_MENUSTYLE
NX_MINIWINDOWSTYLE
NX_MINIWORLDSTYLE
NX_TOKENSTYLE

However, you'd generally choose from the first three styles in this list. Menu styles are appropriate for windows created with methods defined in the Menu class; miniwindows, miniworld icons, and tokens (application icons) are created for you by the Application Kit.

The third argument, *bufferingType*, specifies one of the three possibilities for buffering the drawing done in the Window:

NX_NONRETAINED
NX_RETAINED
NX_BUFFERED

The fourth argument, *mask*, specifies the controls in the Window's title bar and frame. You build the mask by joining (with the bitwise OR operator) the individual masks for each type of button:

NX_CLOSEBUTTONMASK
NX_RESIZEBUTTONMASK
NX_MINIATURIZEBUTTONMASK

You can get all three controls by using the NX_ALLBUTTONS mask. Although called a "button," NX_RESIZEBUTTONMASK refers to the resize bar. All Windows with a style of NX_RESIZEBARSTYLE must set this mask in order for the resize bar to work.

The fifth argument, *flag*, determines whether or not the Window Server will create a window for the new object immediately. If *flag* is YES, it will defer creating the window until it is ordered on-screen. All display messages sent to the Window or its Views will be 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 Server.

The Window creates a direct instance of the View class to be its default content view. You can replace it with your own object by using the **setContentView:** method.

See also: **– orderFront:**, **– setTitle:**, **– setOneShot:**

initContent:style:backing:buttonMask:defer:screen:

– **initContent:**(const NXRect *)*contentRect*
        style:(int)*aStyle*
        backing:(int)*bufferingType*
        buttonMask:(int)*mask*
        defer:(BOOL)*flag*
        screen:(const NXScreen *)*aScreen*

Initializes the Window object immediately after it has been allocated (by Object's **alloc** or **allocFromZone:** method), and returns **self**. This method is equivalent to **initContent:style:backing:buttonMask:defer:**, except that the content rectangle is specified relative to the lower left corner of *aScreen*.

If *aScreen* is NULL, 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: – **initContent:style:backing:buttonMask:defer:**

invalidateCursorRectsForView:

– **invalidateCursorRectsForView:***aView*

Marks the Window as having invalid cursor rectangles. If the Window is the key window, the Application object will send it a **resetCursorRects** message to have it fix its cursor rectangles before getting the next event. If the Window isn't the key window, it will receive the message when it next becomes the key window. Returns **self**.

See also: – **resetCursorRects**

isDisplayEnabled

– (BOOL)**isDisplayEnabled**

Returns YES if the display methods are currently able to display Views in the receiving Window's view hierarchy, and NO if they're not.

See also: – **disableDisplay**, – **reenableDisplay**, – **display:::** (View)

isDocEdited

– (BOOL)**isDocEdited**

Returns YES if the Window's document has been edited, otherwise returns NO.

See also: – **setDocEdited:**

isExcludedFromWindowsMenu

– (BOOL)**isExcludedFromWindowsMenu**

Returns YES if the Window will not be listed in the application's Windows menu, and NO if it will be.

See also: – **setExcludedFromWindowsMenu:**

isKeyWindow

– (BOOL)**isKeyWindow**

Returns YES if the receiving Window is currently the key window, and NO if it isn't.

See also: – **isMainWindow**, – **becomeKeyWindow**, – **resignKeyWindow**

isMainWindow

– (BOOL)**isMainWindow**

Returns YES if the receiving Window is currently the main window, and NO if it isn't.

See also: – **isKeyWindow**, – **becomeMainWindow**, – **resignMainWindow**

isOneShot

– (BOOL)**isOneShot**

Returns YES if the physical window that the Window object manages is freed when it's removed from the screen list, and NO if not. The default is NO.

See also: – **setOneShot:**

isVisible

– (BOOL)**isVisible**

Returns YES if the Window is in the Window Server's screen list, and NO if it's not. A Window can be in the list and still not be visible, either because it's positioned off-screen or because it's covered by other Windows. In either of these cases, **isVisible** may, nevertheless, return YES.

See also: – **getVisibleRect:** (View)

knowsPagesFirst:last:

– (BOOL)**knowsPagesFirst:**(int *)*firstPageNum* **last:**(int *)*lastPageNum*

Implemented by subclasses to indicate whether the Window knows where its own pages lie. This method is invoked when printing (or faxing) the Window. Although it can be implemented in a Window subclass, it should not be used in program code.

If this method returns YES, the Window will receive **getRect:forPage:** messages querying it for the rectangles corresponding to specific pages. If it returns NO, pagination will be done automatically. By default, it returns NO.

Just before this method is invoked, the first page to be printed is set to 1 and the last page to be printed is set to the maximum integer size. An implementation of this method can set *firstPageNum* to a different initial page (for example, a chapter may start on page 40), even if it returns NO. If it returns YES, *lastPageNum* can be set to a different final page. If it doesn't reset *lastPageNum*, the subclass implementation of **getRect:forPage:** must be able to signal that a page has been asked for beyond what is available in the document.

See also: – **getRect:forPage:**, – **printPSCode:**

makeFirstResponder:

– **makeFirstResponder:***aResponder*

Makes *aResponder* the first receiver of keyboard events and action messages sent to the Window. If *aResponder* isn't already the Window's first responder, this method first sends a **resignFirstResponder** message to the object that currently is, and a **becomeFirstResponder** message to *aResponder*. However, if the old first responder refuses to resign, no changes are made.

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* should be a Responder of one type or another; it will usually be a View in the Window's view hierarchy.

If successful in making *aResponder* the first responder, this method returns **self**. If not (if the old first responder refuses to resign), it returns **nil**.

See also: – **becomeFirstResponder** (Responder), – **resignFirstResponder** (Responder)

makeKeyAndOrderFront:

– **makeKeyAndOrderFront:***sender*

Moves the Window to the front of the screen list and makes it the key window. This method can be used in action message. It's a shorthand for:

```
[receiver orderWindow:NX_ABOVE relativeTo:0];
[receiver makeKeyWindow];
```

Returns **self**.

See also: – **orderFront:**, – **orderBack:**, – **orderOut:**, – **orderWindow:relativeTo:**

makeKeyWindow

– **makeKeyWindow**

Makes the receiving Window object the key window, and returns **self**.

See also: – **becomeKeyWindow**, – **isKeyWindow**

miniaturize:

– **miniaturize:***sender*

Removes the Window from the screen list and displays its miniwindow counterpart on-screen. If the Window doesn't have a miniwindow counterpart, one is created.

A **miniaturize:** message is generated when the user clicks the miniaturize button in the Window's title bar. This method has a *sender* argument so that it can be used in an action message from a Control. It ignores this argument. Returns **self**.

See also: – **deminiaturize:**

miniwindowIcon

– (const char *)**miniwindowIcon**

Returns the name of the icon that's displayed on the Window's miniwindow counterpart.

See also: – **setMiniwindowIcon:**

moveTo::

– **moveTo:**(NXCoord)*x* **:**(NXCoord)*y*

Repositions the Window on the screen. The arguments specify the new location of the window—the lower left corner of its frame rectangle—in screen coordinates. Returns **self**.

See also: – **dragFrom::eventNum:**, – **moveTopLeftTo::**

moveTo::screen:

– **moveTo:**(NXCoord)*x* **:**(NXCoord)*y* **screen:**(const NXScreen *)*aScreen*

Repositions the Window so that its lower left corner lies at (*x*, *y*) relative to a coordinate origin at the lower left corner of *aScreen*. If *aScreen* is NULL, this method is the same as **moveTo::**. Returns **self**.

moveTopLeftTo::

– **moveTopLeftTo:**(NXCoord)*x* **:**(NXCoord)*y*

Repositions the Window on the screen. The arguments specify the new location of the Window's top left corner—the top left corner of its frame rectangle—in screen coordinates. Returns **self**.

See also: – **dragFrom::eventNum:**, – **moveTo::**

moveTopLeftTo::screen:

– **moveTopLeftTo:**(NXCoord)*x* **:**(NXCoord)*y* **screen:**(const NXScreen *)*aScreen*

Repositions the Window so that its top left corner lies at (*x*, *y*) relative to a coordinate origin at the lower left corner of *aScreen*. If *aScreen* is NULL, this method is the same as **moveTopLeftTo::**. Returns **self**.

See also: – **moveTo::**

openSpoolFile:

– **openSpoolFile:**(char *)*filename*

Opens the *filename* file for print spooling. This method is invoked when printing (or faxing) the Window; it shouldn't be used in program code. However, you can override it to modify its behavior.

If *filename* is NULL or an empty string (*filename*[0] is '\0'), PostScript code for the Window will be sent directly to the printing daemon, **npd**, without opening a file. (However, if the Window is being previewed or saved, a default file is opened in /**tmp**).

If a *filename* is provided, the file is opened. The printing machinery will then write the PostScript code to that file and the file will be printed using **lpr**.

This method opens a Display PostScript context that will write to the spool file, and sets the context of the global PrintInfo object to this new context. It returns **nil** if the file can't be opened.

See also: – **printPSCode:**

orderBack:

– **orderBack:***sender*

Moves the Window to the back of its tier in the screen list. It may also change the key window and main window. This method is a shorthand for:

```
[receiver orderWindow:NX_BELOW relativeTo:0];
```

Returns **self**.

See also: – **orderFront:**, – **orderOut:**, – **orderWindow:relativeTo:**, – **makeKeyAndOrderFront:**

orderFront:

– **orderFront:***sender*

Moves the Window to the front of the screen list. It may also change the key window and main window. This method is a shorthand for:

```
[receiver orderWindow:NX_ABOVE relativeTo:0];
```

Returns **self**.

See also: – **orderBack:**, – **orderOut:**, – **orderWindow:relativeTo:**, – **makeKeyAndOrderFront:**

orderOut:

– **orderOut:***sender*

Takes the Window out of the screen list. It may also change the key window and main window. This method is a shorthand for:

```
[receiver orderWindow:NX_OUT relativeTo:0];
```

Returns **self**.

See also: – **orderFront:**, – **orderBack:**, – **orderWindow:relativeTo:**

orderWindow:relativeTo:

– **orderWindow:**(int)*place* **relativeTo:**(int)*otherWin*

Repositions the window in the Window Server's screen list. *place* can be one of:

NX_ABOVE
NX_BELOW
NX_OUT

If it's NX_OUT, the window is removed from the screen list and *otherWin* is ignored. If it's NX_ABOVE or NX_BELOW, *otherWin* is the window number of the window that the receiving Window is to be placed above or below. If *otherWin* is 0, the receiving Window will be placed above or below all other windows. Returns self.

See also: – **orderFront:**, – **orderBack:**, – **orderOut:**, – **makeKeyAndOrderFront:**

performClose:

– **performClose:***sender*

Simulates the user clicking the close button by momentarily highlighting the button then closing the window. Returns **self**.

See also: – **performClick:** (Button), – **close**, – **performMiniaturize:**

performMiniaturize:

– **performMiniaturize:***sender*

Simulates the user clicking the miniaturize button by momentarily highlighting the button then miniaturizing the window. Returns **self**.

See also: – **performClick:** (Button), – **miniaturize:**, – **performClose:**

placePrintRect:offset:

– **placePrintRect:**(const NXRect *)*aRect* **offset:**(NXPoint *)*location*

Determines the location of the rectangle being printed on the physical page. This method is invoked when printing (or faxing) the Window; it should not be used in program code. However, you can override it to change the way it places the rectangle.

aRect specifies the rectangle being printed on the current page; *location* is set by this method to be the offset of the rectangle from the lower left corner of the page. All coordinates are in the base coordinate system (that of the page itself).

By default, if the flags for centering are YES in the global PrintInfo object, this method centers the rectangle within the margins. If the flags are NO, it abuts the rectangle against the top and left margins.

See also: – **getRect:forPage:**, – **printPSCode:**

placeWindow:

– **placeWindow:**(const NXRect *)*frameRect*

Resizes the window without redrawing any of its contents. *frameRect* specifies a structure that contains the new frame rectangle of the window in screen coordinates. The rectangle encloses the entire window, including the border, title bar, and resize bar.

This method allows resizing from any window corner or from any point along the window border, but it doesn't move what's displayed within the window or alter the origin of the base coordinate system. Returns **self**.

See also: – **sizeWindow::**, – **moveTo::**, – **placeWindowAndDisplay:**

placeWindow:screen

– **placeWindow:**(const NXRect *)*frameRect* **screen:**(const NXScreen *)*aScreen*

Resizes the window, just as **placeWindow:** does, except that the frame rectangle is specified relative to a coordinate origin at the lower left corner of *aScreen*. If *aScreen* is NULL, this method is the same as **placeWindow:**. Returns **self**.

See also: – **placeWindow:**, – **placeWindowAndDisplay:**

placeWindowAndDisplay:

– **placeWindowAndDisplay:**(const NXRect *)*frameRect*

Resizes the window, just as **placeWindow:** does, but redisplays its contents before the resized window is shown to the user. This prevents the resized window (with unaltered contents) from being displayed before the Views that draw within the window are given a change to adjust to its new size. Returns **self**.

See also: – **placeWindow:**

printPSCode:

– **printPSCode:***sender*

Prints the Window (all the Views in its view hierarchy including the frame view). A return value of **nil** indicates that there were errors in generating the PostScript code or that the user canceled the job.

This method normally brings up the Print panel before actually beginning printing. But if *sender* implements a **shouldRunPrintPanel:** method, that method will be invoked to first query whether to run the panel. If **shouldRunPrintPanel:** returns NO, the Print panel won't be displayed, and the Window will be printed using the last settings of the panel.

See also: – **smartPrintPSCode:**, – **faxPSCode:**, – **shouldRunPrintPanel:** (Object Methods)

read:

– **read:**(NXTypedStream *)*stream*

Reads the Window and its Views from the typed stream *stream.*

See also: – **write:**

reenableDisplay

– **reenableDisplay**

Counters the effect of **disableDisplay**, reenabling the display methods defined in the View class to display Views located within the Window. Returns **self**.

See also: – **disableDisplay**, – **isDisplayEnabled**, – **display:::** (View)

reenableFlushWindow

– **reenableFlushWindow**

Reenables the **flushWindow** method for the Window after it was disabled through a previous **disableFlushWindow** message. Returns **self**.

See also: – **disableFlushWindow**, – **flushWindow**

removeCursorRect:cursor:forView:

– **removeCursorRect:**(const NXRect *)*aRect*
  cursor:*anObj*
  forView:*aView*

Invoked by View's **removeCursorRect:cursor:** method. Do not use this method; use **removeCursorRect:cursor:** instead.

See also: – **removeCursorRect:cursor:** (View), – **resetCursorRects** (View)

removeFromEventMask:

– (int)**removeFromEventMask:**(int)*oldEvents*

Removes the event types specified by *oldEvents* from the Window's event mask, and returns the old mask.

This method is typically used when an object sets up its own modal event loop to respond to certain events. The return value should be used to restore the Window's original event mask when the modal loop is done.

See also: – **eventMask**, – **setEventMask:**, – **addToEventMask:**

resetCursorRects

– **resetCursorRects**

Removes all existing cursor rectangles from the Window, then recreates the cursor rectangles by sending a **resetCursorRects** message to every View in the Window's view hierarchy. Returns **self**.

This method is typically invoked by the Application object when it detects that the key window's cursor rectangles are invalid. In program code, it's more efficient to send a **invalidateCursorRectsForView:** message to fix incorrect cursor rectangles, rather than **resetCursorRects**.

See also: – **invalidateCursorRectsForView:**, – **resetCursorRects** (View)

resignKeyWindow

– **resignKeyWindow**

Records the fact that the receiver is no longer the key window, then passes the **resignKeyWindow** message on to the first responder, if the first responder can respond. The Window's delegate is sent a **windowDidResignKey:** message, if it can respond. Returns **self**.

The Application object sends a **resignKeyWindow** message to the current key window whenever another Window is about to be made the new key window.

If you define a Window subclass and implement your own version of this method, it should include a message to **super** to perform this version as well.

See also: – **becomeKeyWindow**, – **resignMainWindow**, – **setDelegate:**

resignMainWindow

– **resignMainWindow**

Records the fact that the receiving Window is no longer the main window, and sends the Window's delegate a **windowDidResignMain:** message to notify it of the change in status, if the delegate can respond. Returns **self**.

The Application object sends a **resignMainWindow** message to the current main window whenever another Window is about to become the new main window.

See also: – **becomeMainWindow**, – **resignKeyWindow**

rightMouseDown:

– **rightMouseDown:**(NXEvent *)*theEvent*

Responds to uncaught right mouse-down events by passing the message on the Application object. By default, a right mouse-down event in a window causes the main menu to pop up under the cursor. Returns the Application object.

See also: – **rightMouseDown:** (Application)

screen

– (const NXScreen *)**screen**

Returns a pointer to the screen that the Window is on. If the Window is partly on one screen and partly on another, the screen where most of it lies is the one returned.

See also: – **bestScreen**

screenChanged:

– **screenChanged:**(NXEvent *)*theEvent*

Responds to a screen-changed subevent (of the kit-defined event) by sending the Window's delegate a **windowDidChangeScreen:** message, if the delegate can respond. If the Window has a dynamic depth limit, this method also changes the depth limit to match the new device.

A screen-changed subevent is generated when the user releases the mouse button after dragging a window partially or all the way onto another screen.

sendEvent:

– **sendEvent:**(NXEvent *)*theEvent*

Dispatches mouse and keyboard events sent to the Window by the Application object. This method is part of the main event loop and should never be invoked in program code.

setBackgroundColor:

– **setBackgroundColor:**(NXColor)*color*

Sets the background color of the Window to *color*. If set, the background color is used in place of the background gray when the Window is on a color screen. Returns **self**.

See also: – **backgroundColor**

setBackgroundGray:

– **setBackgroundGray:**(float)*value*

Sets the background gray of the Window. *value* should lie in the range 0.0 (black) to 1.0 (white). To obtain pure shades of gray, use one of the following constants:

NX_BLACK
NX_DKGRAY
NX_LTGRAY
NX_WHITE

Returns **self**.

See also: – **backgroundGray**

setContentView:

– **setContentView:***aView*

Makes *aView* the Window's content view after removing the former content view from the Window's view hierarchy. *aView* is resized so that it exactly fills the content area of the Window; its **superview**, **nextResponder**, and **window** instance variables are altered to reflect its new status. This method returns the **id** of the former content view so that you can free it or assign it another position in a view hierarchy. Once the content view is set, you should not attempt to change its frame rectangle by sending it a **setFrame:**, **moveTo::**, **sizeTo::**, or other message. The content view's frame is reset by the Window whenever the window is resized.

See also: – **contentView**

setDelegate:

– **setDelegate:***anObject*

Makes *anObject* the Window's delegate, and returns **self**. The delegate is given a chance to respond to action messages that work their way up the responder chain to the Window (through Application's **sendAction:to:from:** method). It can also respond to notification messages sent by the Window. See "METHODS IMPLEMENTED BY THE DELEGATE" near the end of this class specification.

See also: – **delegate**, – **tryToPerform:with:**, – **sendAction:to:from:** (Application)

setDepthLimit:

– **setDepthLimit:**(NXWindowDepth)*limit*

Sets the depth limit of the Window to *limit*, which should be one of the following enumerated values (defined in the header file **appkit/graphics.h**):

NX_TwoBitGrayDepth
NX_EightBitGrayDepth
NX_TwelveBitRGBDepth
NX_TwentyFourBitRGBDepth

Returns **self**.

See also: – **depthLimit**, + **defaultDepthLimit**, – **setDynamicDepthLimit:**

setDocEdited:

– **setDocEdited:**(BOOL)*flag*

Sets whether or not the document displayed in the Window has been edited but not saved. If *flag* is YES, the Window's close button will display a broken "X" to indicate that the document needs to be saved. If *flag* is NO, the close button will be shown with a solid "X". The default is NO. Returns **self**.

See also: – **isDocEdited**

setDynamicDepthLimit:

– **setDynamicDepthLimit:**(BOOL)*flag*

Sets whether the Window's depth limit should change to match the depth of the display device that it's on. If *flag* is YES, the depth limit will depend on which screen the Window is on. If *flag* is NO, the Window will have the default depth limit. A different, and nondynamic, depth limit can be set with the **setDepthLimit:** method. Returns **self**.

See also: – **hasDynamicDepthLimit**, + **defaultDepthLimit**, – **setDepthLimit:**

setEventMask:

– (int)**setEventMask:**(int)*newMask*

Assigns a new event mask to the Window and returns the original event mask. The mask tells the Window Server which types of events the Window wants to receive. It's formed by joining the masks for individual events using the bitwise OR operator. The constants for individual event masks are listed below. Those that are included in the default event mask for a Window are marked with an asterisk.

```
*  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_CURSORUPDATEMASK
   NX_JOURNALEVENTMASK
   NX_NULLEVENTMASK
```

Miniwindows and application icons have the same default event mask as other Windows, except that all keyboard events are excluded. The default mask for a Menu includes only left and right mouse-down, mouse-up, and mouse-dragged events and the kit-defined event.

See also: – **eventMask**, – **addToEventMask:**, – **removeFromEventMask:**

setExcludedFromWindowsMenu:

– **setExcludedFromWindowsMenu:**(BOOL)*flag*

Sets whether the Window will be excluded from the Windows menu. If *flag* is YES, it won't be listed in the menu. If *flag* is NO, it will be listed when it or its miniwindow is on-screen. The default is NO. Returns **self**.

See also: – **isExcludedFromWindowsMenu**

setFreeWhenClosed:

– **setFreeWhenClosed:**(BOOL)*flag*

Determines the Window's behavior when it receives a **close** message. If *flag* is NO, the Window is just hidden (taken out of the screen list). If *flag* is YES, the Window is hidden and then freed. The default for Windows is YES; the default for Panels and Menus is NO. Returns **self**.

See also: – **close**, – **free**

setHideOnDeactivate:

– **setHideOnDeactivate:**(BOOL)*flag*

Determines whether the Window will disappear when the application is inactive. If *flag* is YES, the Window is hidden (taken out of the screen list) when the application stops being the active application. If *flag* is NO, the Window stays on-screen. The default for Windows is NO; the default for Panels and Menus is YES. Returns **self**.

See also: – **doesHideOnDeactivate:**

setMiniwindowIcon:

– **setMiniwindowIcon:**(const char *)*name*

Sets the icon to be used during window miniaturization. There is a 48-by-48 pixel area available on a miniaturized window for displaying an icon. The NXImage class will look in the __ICON, __EPS, and __TIFF segments of the application executable to create the icon upon miniaturization if it's not already available.

See also: – **miniwindowIcon**, – **windowWillMiniaturize:toMiniwindow:**

setOneShot:

– **setOneShot:**(BOOL)*flag*

Sets whether the physical window that the Window object manages should be freed when it's removed from the screen list (and another one created if it's returned to the screen). This is appropriate behavior for windows that the user might use once or twice but not display continually. The default is NO. Returns **self**.

See also: – **isOneShot**

setTitle:

– **setTitle:**(const char *)*aString*

Changes the string that appears in the Window's title bar to *aString*. You don't have to redisplay the Window to make the new title appear. Returns **self**.

See also: – **title**, – **setTitleAsFilename:**

setTitleAsFilename:

– **setTitleAsFilename:**(const char *)*aString*

Sets *aString* to be the title of the Window, but formats it as a pathname to a file. The file name is displayed first, followed by an em dash and the directory path. The em dash is offset by two spaces on either side. For example:

```
MyFile  —  /Net/server/group/home
```

The string can be a full or relative pathname. If it lacks any '/' characters, it won't be formatted.

Returns **self**.

See also: **– title**, **– setTitle:**

setTrackingRect:inside:owner:tag:left:right:

– **setTrackingRect:**(const NXRect *)*aRect*
  inside:(BOOL)*insideFlag*
  owner:*anObject*
  tag:(int)*trackNum*
  left:(BOOL)*leftDown*
  right:(BOOL)*rightDown*

Sets up a tracking rectangle in the Window through the **settrackingrect** operator. The first argument, *aRect*, is a pointer to the tracking rectangle and is specified in the Window's current coordinate system. The second argument, *insideFlag*, indicates whether the cursor starts off inside the rectangle (YES) or outside it (NO). The third argument, *anObject*, is the **id** of the object, usually a View or an NXCursor, that will handle the mouse-entered and mouse-exited events that are generated for the rectangle; the Application object dispatches these events directly to the responsible object. The fourth argument, *trackNum*, is a number that you assign to identify the rectangle.

If *leftDown* is YES, the Window Server will generate mouse-entered and mouse-exited events for the rectangle only while the left mouse button is down; if *rightDown* is YES, events are generated only while the right button is down.

Returns **self**.

See also: **– discardTrackingRect:**

sizeWindow::

– **sizeWindow:**(NXCoord)*width* **:**(NXCoord)*height*

Resizes the window so that its content area has the specified *width* and *height* in base coordinates. The lower left corner of the window remains constant. Returns **self**.

See also: – **placeWindow:**

smartFaxPSCode:

– **smartFaxPSCode:***sender*

Prints the Window (all the Views in its view hierarchy including the frame view) to a fax modem so that it will fit on a single sheet of paper. This method tries to set up the various parameters of the printing machinery to create a pleasing result. The image is centered horizontally and vertically, and the orientation of the paper (portrait or landscape) is set to match the dimensions of the window. These settings are temporary, however, and do not permanently affect the global PrintInfo object.

In the current user interface, faxing is initiated from within the Print panel. However, with this method, you can provide users with an independent control for faxing a Window.

This method normally brings up the Fax panel before actually beginning printing. But if *sender* implements a **shouldRunPrintPanel:** method, that method will be invoked to first query whether to run the panel. If **shouldRunPrintPanel:** returns NO, the Fax panel won't be displayed, and the Window will be printed using the last settings of the panel.

A return value of **nil** indicates that there were errors in generating the PostScript code or that the user canceled the job.

See also: – **faxPSCode:**, – **smartPrintPSCode:**, – **shouldRunPrintPanel:** (Object Methods)

smartPrintPSCode:

– **smartPrintPSCode:***sender*

Prints the Window (all the Views in its view hierarchy including the frame view) on a single sheet of paper. This method tries to set up the various parameters of the printing machinery to create a pleasing result. The image is centered horizontally and vertically, and the orientation of the paper (portrait or landscape) is set to match the dimensions of the window. These settings are temporary, however, and do not permanently affect the global PrintInfo object.

This method normally brings up the Print panel before actually beginning printing. But if *sender* implements a **shouldRunPrintPanel:** method, that method will be invoked to first query whether to run the panel. If **shouldRunPrintPanel:** returns NO, the Print panel won't be displayed, and the Window will be printed using the last settings of the panel.

A return value of **nil** indicates that there were errors in generating the PostScript code or that the user canceled the job.

See also: – **printPSCode:**, – **smartFaxPSCode:**, – **shouldRunPrintPanel:** (Object Methods)

spoolFile:

– **spoolFile:**(const char *)*filename*

Spools the generated PostScript code in *filename* to the printer. This method is invoked automatically when printing (or faxing) the Window.

See also: – **openSpoolFile:**

style

– (int)**style**

Returns one of several values, indicating the Window's style:

NX_PLAINSTYLE
NX_TITLEDSTYLE
NX_RESIZEBARSTYLE
NX_MENUSTYLE
NX_MINIWINDOWSTYLE
NX_MINIWORLDSTYLE
NX_TOKENSTYLE

See also: – **initContent:style:backing:buttonMask:defer:**

title

– (const char *)**title**

Returns the string that appears in the title bar of the window. If the title was formatted by the **setTitleAsFilename:** method, the formatted string is returned.

See also: – **setTitle:**, – **setTitleAsFilename:**

tryToPerform:with:

– (BOOL)**tryToPerform:**(SEL)*anAction* **with:***anObject*

Overrides Responder's version of **tryToPerform:with:** to give the Window's delegate a chance to respond to the action message. If successful in finding a receiver that accepts the *anAction* message (that doesn't return **nil**), this method returns YES. Otherwise, it returns NO.

See also: – **tryToPerform:with:** (Responder)

update

– **update**

Implemented by subclasses to automatically update the Window and redisplay it. Returns **self**.

A Window receives a **update** message:

- After each event, if the Window is in the screen list and the Application object has been instructed to automatically update all Windows. The Application object sends an **update** message to every visible Window after each event has been handled in the main event loop.
- Before the Window is placed in the screen list.
- Before the Window receives a **commandKey:** message.

The message gives the Window a chance to make any changes in its state or display that are contingent on the way an event was handled.

Window's default version of the **update** method sends the delegate a **windowDidUpdate:** message, if the delegate can respond. Subclass versions of the method should send a message to **super** to incorporate Window's version *after* completing the update and just before returning. The Menu class implements this method to disable and enable menu commands as appropriate.

See also: – **updateWindows** (Application), – **setAutoupdate:** (Application)

useOptimizedDrawing:

– **useOptimizedDrawing:**(BOOL)*flag*

Informs the Window whether to optimize focusing and drawing when Views are displayed. The optimizations may prevent sibling subviews from being displayed in the correct order. This matters only if the subviews overlap. Always set *flag* to YES if there are no overlapping subviews within the Window. The default is NO. Returns **self**.

validRequestorForSendType:andReturnType:

– **validRequestorForSendType:**(NXAtom)*typeSent*
  andReturnType:(NXAtom)*typeReturned*

Passes this message on to the Window's delegate, if the delegate can respond (and isn't a Responder with its own next responder). If the delegate can't respond or returns **nil**, this method passes the message to the Application object. If the Application object returns **nil**, this method also returns **nil**, indicating that no object was found that could supply *typeSent* data for a remote message from the Services menu and accept back *typeReturned* data. If such an object was found, it is returned.

Messages to perform this method are initiated by the Services menu. It's part of the mechanism that passes **validRequestorForSendType:andReturnType:** messages up the responder chain.

See also: – **validRequestorForSendType:andReturnType:** (Responder and Application)

widthAdjustLimit

– (float)**widthAdjustLimit**

Returns the fraction of a page that can be pushed onto the next page to prevent items from being cut in half. The limit applies to horizontal pagination. By default, it's 0.2.

This method is invoked during automatic pagination when printing (or faxing) the Window; it should not be used in program code. However, you can override it to return a different value. The value returned should lie between 0.0 and 1.0 inclusive.

See also: – **heightAdjustLimit**

windowExposed:

– **windowExposed:**(NXEvent *)*theEvent*

Responds to a window-exposed event by displaying the portion of the window that the event record indicates should be redrawn, and informing the delegate through a **windowDidExpose:** message. Returns **self**.

See also: – **display::** (View), – **setDelegate:**

windowMoved:

– **windowMoved:**(NXEvent *)*theEvent*

Responds to a window-moved event by recording the new location of the window, and informing the Window delegate through a **windowDidMove:** message. Returns **self**.

If you define a Window subclass and implement your own version of this method, it should include a message to **super** to apply this version as well.

See also: – **dragFrom::eventNum:**, – **setDelegate:**

windowNum

– (int)**windowNum**

Returns the window number of the window corresponding to the receiving Window object. If the Window object doesn't currently have a window, the return value will be equal to or less than 0.

See also: – **initContent:style:backing:buttonMask:defer:**, – **setOneShot:**

windowResized:

– **windowResized:**(NXEvent *)*theEvent*

Responds to a window-resized event by recording the new dimensions of the window and causing it to redisplay. Returns **self**.

Window-resized events are not real events; they're not placed in the event queue. The frame view sends the Window object a **windowResized:** message after the user has resized the window from the resize bar. While the window is being resized, the Window's delegate receives repeated **windowWillResize:toSize:** messages giving it the opportunity to constrain the future size of the window. After the resizing is completed, this **windowResized:** method sends the delegate a **windowDidResize:** message.

See also: – **display::** (View)

worksWhenModal

– (BOOL)**worksWhenModal**

Returns whether the Window is able to receive keyboard and mouse events even when there's a modal panel (an attention panel) on-screen. The default is NO. Only Panels should change this default.

See also: – **setWorksWhenModal:** (Panel)

write:

– **write:**(NXTypedStream *)*stream*

Writes the receiving Window to the typed stream *stream*, along with its content view and miniwindow counterpart. The delegate and field editor are not explicitly written, but all subviews of the content view will be.

See also: – **read:**

METHODS IMPLEMENTED BY THE DELEGATE

windowDidBecomeKey:

– **windowDidBecomeKey:***sender*

Responds to a message informing the delegate that the *sender* Window has just become the key window.

See also: – **becomeKeyWindow**

windowDidBecomeMain:

– **windowDidBecomeMain:***sender*

Responds to a message informing the delegate that the *sender* Window has just become the main window.

See also: – **becomeMainWindow**

windowDidChangeScreen:

– **windowDidChangeScreen:***sender*

Responds to a message informing the delegate that the *sender* Window has received a screen-changed subevent (of the kit-defined event).

See also: – **screenChanged:**

windowDidDeminiaturize:

– **windowDidDeminiaturize:***sender*

Responds to a message informing the delegate that the user has double-clicked the *sender* Window's miniwindow counterpart, returning the Window to the screen and hiding the miniwindow.

See also: – **deminiaturize:**, – **windowDidMiniaturize:**

windowDidExpose:

– **windowDidExpose:***sender*

Responds to a message informing the delegate that the *sender* Window received a window-exposed subevent of the kit-defined event.

See also: – **windowExposed:**

windowDidMiniaturize:

– **windowDidMiniaturize:***sender*

Responds to a message informing the delegate that the user has miniaturized the *sender* Window.

See also: – **windowWillMiniaturize:toMiniwindow:**, – **windowDidDeminiaturize:**

windowDidMove:

– **windowDidMove:***sender*

Responds to a message informing the delegate that the user moved the *sender* Window.

See also: – **windowMoved:**

windowDidResignKey:

– **windowDidResignKey:***sender*

Responds to a message informing the delegate that the *sender* Window is no longer the key window.

See also: – **resignKeyWindow**

windowDidResignMain:

– **windowDidResignMain:***sender*

Responds to a message informing the delegate that the *sender* Window is no longer the main window.

See also: – **resignMainWindow**

windowDidResize:

– **windowDidResize:***sender*

Responds to a message informing the delegate that the user has finished resizing the *sender* Window. The new size of the Window can be obtained by sending it a **getFrame:** message.

See also: – **windowWillResize:toSize:**, – **getFrame:**

windowDidUpdate:

– **windowDidUpdate:***sender*

Responds to a message that's sent when the *sender* Window receives an **update** message.

See also: – **update**

windowWillClose:

– **windowWillClose:***sender*

Responds to a message informing the delegate that the *sender* Window is about to close. If the delegate returns **nil**, the Window won't close.

windowWillMiniaturize:toMiniwindow:

– **windowWillMiniaturize:***sender* **toMiniwindow:***miniwindow*

Responds to a message informing the delegate that the user will miniaturize the *sender* Window. The delegate can install a special content View for miniwindow, or set its title. The default title is the same as *sender*'s.

See also: – **windowDidMiniaturize:**, – **miniaturize:**

windowWillResize:toSize:

– **windowWillResize:***sender* **toSize:**(NXSize *)*frameSize*

Responds to a message informing the delegate that the user is trying to resize the *sender* Window. During window resizing, the delegate is sent continuous **windowWillResize:toSize:** messages as the user drags the window's outline. The second argument, *frameSize*, is a a pointer to an NXSize structure containing the size (in screen coordinates) that the window will be resized to. If the delegate wants to constrain the window size, it may update the structure to the desired size. The window outline is displayed at the constrained size provided by the delegate.

See also: – **windowDidResize:**, – **windowResized:**

windowWillReturnFieldEditor:toObject:

– **windowWillReturnFieldEditor:***sender* **toObject:***client*

Responds to a message informing the delegate that *client* has requested the *sender* Window's field editor, the Text object that performs various editing tasks within the Window. If the delegate's implementation of this method returns an object other than **nil**, the Window substitutes it for the field editor and returns it to *client*.

See also: – **getFieldEditor:for:**

CONSTANTS AND DEFINED TYPES

```
/*
 * Window styles
 */
#define NX_PLAINSTYLE           0
#define NX_TITLEDSTYLE          1
#define NX_MENUSTYLE            2
#define NX_MINIWINDOWSTYLE      3
#define NX_MINIWORLDSTYLE       4
#define NX_TOKENSTYLE           5
#define NX_RESIZEBARSTYLE       6
#define NX_FIRSTWINSTYLE        NX_PLAINSTYLE
#define NX_LASTWINSTYLE         NX_RESIZEBARSTYLE
#define NX_NUMWINSTYLES \
     (NX_LASTWINSTYLE - NX_FIRSTWINSTYLE + 1)

/*
 * Control masks
 */
#define NX_CLOSEBUTTONMASK              1
#define NX_RESIZEBUTTONMASK             2
#define NX_MINIATURIZEBUTTONMASK        4
#define NX_ALLBUTTONS \
     (NX_CLOSEBUTTONMASK|NX_RESIZEBUTTONMASK|NX_MINIATURIZEBUTTONMASK)

/*
 * Sizes of icon images and windows
 */
#define NX_ICONWIDTH            48.0
#define NX_ICONHEIGHT           48.0
#define NX_TOKENWIDTH           64.0
#define NX_TOKENHEIGHT          64.0
```

```
/*
 * Window tiers
 */
#define NX_NORMALLEVEL          0
#define NX_FLOATINGLEVEL        3
#define NX_DOCKLEVEL            5
#define NX_SUBMENULEVEL         10
#define NX_MAINMENULEVEL        20
```

Chapter 3
C Functions

Chapter 3
C Functions

This chapter gives detailed descriptions of the C functions provided by NeXT. Also included here are some macros that behave like functions. For this chapter, the functions and macros are divided into two groups:

- NeXTstep, which includes functions and macros defined in the Application Kit, functions for using streams and typed streams, and Display PostScript functions
- Run-time functions for the Objective-C language

Within these divisions, functions are subgrouped with other functions that perform related tasks. These subgroups are described in alphabetical order by the name of the first function listed in the subgroup. Functions within subgroups are also listed alphabetically, with a pointer to the subgroup's description.

For convenience, these functions are summarized in the *NeXT Technical Summaries* manual. The summary lists functions by the same subgroups used in this chapter and combines several related subgroups under a heading such as "Rectangle Functions" or "Stream Functions." The calling sequence for each function is shown in the summary.

Note that under the SYNOPSIS heading in the function descriptions, the lowest-level header file is specified in the **#import** statement; you might instead include a header file like **appkit/appkit.h**, which includes many other, lower-level header files. For details on these files, see Chapter 1, "Data Types and Constants."

NeXTstep Functions

This section contains descriptions of two types of functions: those that implement NeXT's system-dependent interface to the Display PostScript system and those that support the various Application Kit classes. The Display PostScript system functions have a "DPS" prefix; the Application Kit functions have an "NX" prefix. The descriptions of the "DPS" functions assume knowledge of the Display PostScript system. For the primary documentation of this system, refer to *Extensions for the Display PostScript System* and *Client Library Reference Manual*, both by Adobe Systems Incorporated. See "Suggested Reading" in *Technical Summaries* for information on Adobe documentation.

DPSAddFD(), DPSRemoveFD()

SUMMARY Add or remove monitored file descriptor

LIBRARY libNeXT_s.a

SYNOPSIS

#import <dpsclient/dpsclient.h>

void **DPSAddFD**(int *fd*, DPSFDProc *handler*, void **userData*, int *priority*)
void **DPSRemoveFD**(int *fd*)

DESCRIPTION

DPSAddFD() adds a file descriptor to the list of those that the client library can check each time it attempts to retrieve an event. The integer *fd* is the file descriptor (as returned by **open()**) to be added. When data can be read from the file identified by *fd*, the function *handler* is called (assuming an appropriate value of *priority*, as explained below). The third argument, *userData*, is a pointer that the application can use to pass some data to *handler.*

The integer *priority* lets you specify the execution priority of *handler.* A priority level can be from 0 to 30. During normal execution of a program based on the Application Kit, the function that returns events from the Window Server will check *fd* if *priority* is NX_BASETHRESHOLD (a value of 1) or higher. When the application displays an attention panel, *fd* is checked only if *priority* is NX_RUNMODALTHRESHOLD (a value of 5) or higher. During a modal event loop, *fd* is checked only if *priority* is NX_MODALRESPTHRESHOLD (a value of 10) or higher.

Note: NX_BASETHRESHOLD, NX_RUNMODALTHRESHOLD, and NX_MODALRESPTHRESHOLD are defined in the header file **appkit/Application.h**.

The function registered as *handler* has the form:

void *func*(int *fd*, void **userData*)

where *fd* is the file descriptor of the file that's ready to be read and *userData* is a reference to the data you specified in the call to **DPSAddFD()**.

DPSRemoveFD() removes the specified file descriptor from the list of those that the application will check.

SEE ALSO

DPSAddPort(), **DPSAddTimedEntry()**

DPSAddPort(), DPSRemovePort()

SUMMARY Add or remove monitored Mach port

LIBRARY libNeXT_s.a

SYNOPSIS

#import <dpsclient/dpsclient.h>

void **DPSAddPort**(port_t *newPort*, DPSPortProc *handler*, int *maxSize*,
    void **userData*, int *priority*)
void **DPSRemovePort**(port_t *port*)

DESCRIPTION

DPSAddPort() adds a Mach port to the list of ports that an application based on the Application Kit can check each time it attempts to retrieve an event. *newPort* identifies the Mach port to be monitored. When a message arrives at the port, the function *handler* is called (assuming an appropriate value of *priority*, as explained below). The type DPSPortProc is defined in the header file **dpsclient/dpsNeXT.h**. The *maxSize* argument declares the maximum size of the in-line data (including the message header) that will be received in the message. The pointer *userData* can be used to pass some data to *handler.*

The integer *priority* lets you specify the execution priority of *handler.* A priority level can be from 0 to 30. During normal execution of a program based on the Application Kit, the function that returns events from the Window Server will check *newPort* if *priority* is NX_BASETHRESHOLD (a value of 1) or higher. When the application displays a modal panel, *newPort* is checked only if *priority* is NX_RUNMODALTHRESHOLD (a value of 5) or higher. During a modal event loop, *newPort* is checked only if *priority* is NX_MODALRESPTHRESHOLD (a value of 10) or higher.

Note: NX_BASETHRESHOLD, NX_RUNMODALTHRESHOLD, and NX_MODALRESPTHRESHOLD are defined in the header file **appkit/Application.h**.

The function registered as *handler* has the form:

void *func*(msg_header_t **msg*, void **userData*)

where *msg* is a pointer to the message that was received at the port and *userData* is a reference to the data you specified in the call to **DPSAddPort()**.

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 message buffer identified by *msg* is overwritten whenever your application gets events, receives values from the Window Server, or receives data from a monitored port. If you want to preserve the message, copy the contents of the message buffer into local storage before you take an action that might overwrite it.

DPSRemovePort() removes the specified port from the list of those that the application will check.

The Application Kit provides an object-oriented interface to Mach ports through the Listener and Speaker classes. To send messages between two applications based on the Application Kit, use Speaker and Listener objects. To monitor a Mach port directly, use **DPSAddPort**().

DPSAddTimedEntry(), DPSRemoveTimedEntry()

SUMMARY Add or remove timed entry

LIBRARY libNeXT_s.a

SYNOPSIS

#import <dpsclient/dpsclient.h>

DPSTimedEntry **DPSAddTimedEntry**(double *period*, DPSTimedEntryProc *handler*, void **userData*, int *priority*)
void **DPSRemoveTimedEntry**(DPSTimedEntry *teNumber*)

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 integer *priority* lets you specify the execution priority of *handler*. A priority level can be from 0 to 30. During normal execution of a program based on the Application Kit, the function that returns events from the Window Server will check if *handler* is due to be called if *priority* is NX_BASETHRESHOLD (a value of 1) or higher. When the application displays a modal panel, *handler* is checked only if *priority* is NX_RUNMODALTHRESHOLD (a value of 5) or higher. During a modal event loop, *handler* is checked only if *priority* is NX_MODALRESPTHRESHOLD (a value of 10) or higher.

Note: NX_BASETHRESHOLD, NX_RUNMODALTHRESHOLD, and NX_MODALRESPTHRESHOLD are defined in the header file **appkit/Application.h**.

The function registered as *handler* has the form:

```
void func(DPSTimedEntry teNumber, double now, char *userData)
```

where *teNumber* 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.

DPSRemoveTimedEntry() removes a previously registered timed entry.

RETURN

DPSAddTimedEntry() returns a number identifying the timed entry or –1 to indicate an error.

DPSCreateContext(), DPSCreateContextWithTimeoutFromZone(), DPSCreateStreamContext()

SUMMARY	Create PostScript execution context
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <dpsclient/dpsclient.h>

DPSContext **DPSCreateContext**(const char **hostName*, const char **serverName*,
DPSTextProc *textProc*, DPSErrorProc *errorProc*)
DPSContext **DPSCreateContextWithTimeoutFromZone**(const char **hostName*,
const char **serverName*, DPSTextProc *textProc*, DPSErrorProc *errorProc*,
int *timeout*, NXZone **zone*)
DPSContext **DPSCreateStreamContext**(NXStream **stream*, int *debugging*,
DPSProgramEncoding *progEnc*, DPSNameEncoding *nameEnc*,
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 well-known name for the Window Server is used. If both arguments are NULL, **DPSCreateContext()** checks to see whether access rights to the Window Server's port have been inherited from the application's parent. (For example, an application launched by the Workspace Manager™ gains a connection to the Window Server by inheriting it from the Workspace Manager.) If the rights weren't inherited from the parent, **DPSCreateContext()** queries the local machine for the Window Server's port using a well-known name.

The last two arguments, *textProc* and *errorProc*, refer to call-back procedures that handle text returned from the Window Server and errors generated on either side of the connection. See "Handling Output" in the *Client Library Reference Manual* by Adobe Systems Incorporated for more details.

For an application that's based on the Application Kit, you could create an additional context by making this call:

```
DPSContext  context;

context = DPSCreateContext(NXGetDefaultValue([NXApp appName],
                "NXHost"),NXGetDefaultValue([NXApp appName],
                "NXPSName"),
                 NULL, NULL);
```

This example queries the application's default values for the indentity 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.

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 (which may have memory, a file, or a Mach port as a destination) rather than to the Window Server. The first argument, *stream*, is a pointer to an NXStream, 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. (See *Extensions for the Display PostScript System* for more information.) The last argument, *errorProc*, identifies the procedure that's called when errors are generated.

Few programmers will need to call either of these functions 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.

RETURN

Each of these functions returns the newly created DPSContext, as defined in the header file **dpsclient/dpsfriends.h**.

EXCEPTIONS

DPSCreateContext() and **DPSCreateContextWithTimeoutFromZone()** raise a dps_err_outOfMemory exception if they encounter difficulty allocating ports or other resources for the new context. They raise a dps_err_cantConnect exception if they can't return a context within the timeout period.

DPSCreateContextWithTimeoutFromZone() → See DPSCreateContext()

DPSCreateStreamContext() → See DPSCreateContext()

DPSDefineUserObject(), DPSUndefineUserObject()

SUMMARY	Return index for top object of operand stack
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <dpsclient/dpsclient.h>

int **DPSDefineUserObject**(int *index*)
void **DPSUndefineUserObject**(int *index*)

DESCRIPTION

DPSDefineUserObject() associates *index* with the PostScript object that's on the top of the operand stack, thereby creating a user object. (See *Extensions for the Display PostScript System* for a description of user objects.) 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.

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()**.

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.

RETURN

DPSDefineUserObject(), if successful in assigning an index, returns the index that the object was assigned. If unsuccessful, it returns 0.

DPSDiscardEvents() → See DPSGetEvent()

DPSDoUserPath(), DPSDoUserPathWithMatrix()

SUMMARY	Send PostScript path to Window Server and execute
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <dpsclient/dpsclient.h>

void **DPSDoUserPath**(void **coords*, int *numCoords*, DPSNumberFormat *numType*, char **ops*, int *numOps*, void **bbox*, int *action*)
void **DPSDoUserPathWithMatrix**(void **coords*, int *numCoords*, DPSNumberFormat *numType*, char **ops*, int *numOps*, void **bbox*, int *action*, float *matrix[6]*)

DESCRIPTION

DPSDoUserPath() and **DPSDoUserPathWithMatrix**() send an encoded user path to the Window Server and then execute the operator specified by *action*. (See "User Paths" in *Extensions for the Display PostScript System* for the primary documentation on user paths.) The two functions are identical except for the *matrix* argument required by **DPSDoUserPathWithMatrix**().

The encoded user path is described by the *coords*, *ops*, and *bbox* arguments. The *bbox* and *coords* arguments specify the encoded user path's data string; the *ops* argument refers to the encoded user path's operator string. The *bbox* argument identifies the operands for the **setbbox** operator, and the *coords* argument identifies the coordinates used by the operators encoded in the operator string. You pass the number of elements in the *coords* and *ops* arguments using the *numCoords* and *numOps* arguments.

The *numType* argument specifies the type of the numbers used in the data string. The header file **dpsclient/dpsNeXT.h** defines these constants for *numType*:

dps_float
dps_long
dps_short

You can also specify 16 and 32-bit fixed-point 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. See "Alternate Language Encodings" in *Extensions for the Display PostScript System* for more information.

The *ops* argument refers to an array encoding the operators that will consume the operands in the data string. 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

The first operands in a user path (as identified by *bbox*) are consumed by the **setbbox** operator; however, including **dps_setbbox** in the operator string is optional. If you don't include it, it will be included for you.

Once the user path has been constructed, the operator specified by *action* is executed. These constants are provided for *action*:

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 optional matrix operand used by the **ustroke**, **inustroke**, and **ustrokepath** operators. If *matrix* is NULL, the argument is ignored.

The following program fragment demonstrates the use of **DPSDoUserPath()** by creating 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. It then strokes the path.

```
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);
```

DPSDoUserPathWithMatrix() → See DPSDoUserPath()

DPSFlush()

SUMMARY Send PostScript code to Window Server

LIBRARY libNeXT_s.a

SYNOPSIS

#import <dpsclient/dpsclient.h>

void **DPSFlush()**

DESCRIPTION

DPSFlush() flushes the application's output buffer, forcing any buffered PostScript code or data to the Window Server. **DPSFlush()** is a cover for **DPSFlushContext(DPSGetCurrentContext())**; for more information about these functions, see their descriptions in the *Client Library Reference Manual.*

DPSGetEvent(), DPSPeekEvent(), DPSDiscardEvents()

SUMMARY	Access data from Window Server
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <dpsclient/dpsclient.h>

int **DPSGetEvent**(DPSContext *context*, NXEvent **anEvent*, int *mask*, double *timeout*, int *threshold*)
int **DPSPeekEvent**(DPSContext *context*, NXEvent **anEvent*, int *mask*, double *timeout*, int *threshold*)
void **DPSDiscardEvents**(DPSContext *context*, int *mask*)

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:**) or the function **NXGetOrPeekEvent()** 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()**. (See the *Client Library Reference Manual* for information on **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. The header file **dpsclient/event.h** defines these constants for *mask*:

Constant	Event Type
NX_KEYDOWNMASK	Key-down
NX_KEYUPMASK	Key-up
NX_FLAGSCHANGEDMASK	Flags-changed
NX_LMOUSEDOWNMASK	Mouse-down, left or only mouse button
NX_LMOUSEUPMASK	Mouse-up, left or only mouse button
NX_RMOUSEDOWNMASK	Mouse-down, right mouse button
NX_RMOUSEUPMASK	Mouse-up, right mouse button
NX_MOUSEMOVEDMASK	Mouse-moved
NX_LMOUSEDRAGGEDMASK	Mouse-dragged, left or only mouse button
NX_RMOUSEDRAGGEDMASK	Mouse-dragged, right mouse button
NX_MOUSEENTEREDMASK	Mouse-entered
NX_MOUSEEXITEDMASK	Mouse-exited
NX_TIMERMASK	Timer
NX_CURSORUPDATEMASK	Cursor-update
NX_KITDEFINEDMASK	Kit-defined
NX_SYSDEFINEDMASK	System-defined
NX_APPDEFINEDMASK	Application-defined
NX_ALLEVENTS	All event types

To check for multiple types of events, you can combine these 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.

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

DPSAddFD(), DPSAddPort(), DPSAddTimedEntry(), DPSPostEvent(), NXGetOrPeekEvent()

DPSNameFromTypeAndIndex()

SUMMARY Provide support for user names

LIBRARY libNeXT_s.a

SYNOPSIS

#import <dpsclient/dpsclient.h>

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. (See "System and user name encodings" in the "Alternate Language Encodings" section of *Extensions for the Display PostScript System* for more information.)

RETURN

This function returns a read-only character string.

DPSPeekEvent() → See DPSGetEvent()

DPSPostEvent()

SUMMARY Post event without involving Window Server

LIBRARY libNeXT_s.a

SYNOPSIS

#import <dpsclient/dpsclient.h>

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 record is posted in front of all other records and so will be the next one your application receives. If *atStart* is FALSE, the record is posted behind all other records and so won't be returned until records that precede it are processed.

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(), DPSPrintErrorToStream()

SUMMARY	Handle errors
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <dpsclient/dpsclient.h>

void **DPSPrintError**(FILE **fp*, const DPSBinObjSeq *error*)
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*. (The type DPSBinObjSeq is defined in the header file **dpsclient/dpsfriends.h**.) **DPSPrintError**() prints the error message to the file identified by *fp*; **DPSPrintErrorToStream**() prints the error message to *stream*. (The NXStream structure is defined in the header file **streams/streams.h**.)

You rarely will need to call these functions directly. However, if you reset the error handler for a PostScript execution context, the new handler you install could use one of these functions to process errors that it receives. See the *Client Library Reference Manual* for more information on error handling.

DPSPrintErrorToStream() → See DPSPrintError()

DPSRemoveFD() → See DPSAddFD()

DPSRemovePort() → See DPSAddPort()

DPSRemoveTimedEntry() → See DPSAddTimedEntry()

DPSSetDeadKeysEnabled()

SUMMARY	Enable or disable dead key processing for a context's events
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <dpsclient/dpsclient.h>

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.

See the *NeXT User's Reference* manual for a listing of the keys that produce accent characters.

DPSSetEventFunc()

SUMMARY Set function that filters events

LIBRARY libNeXT_s.a

SYNOPSIS

#import <dpsclient/dpsclient.h>

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.

The following declaration is provided in the header file **dpsclient/dpsNeXT.h** for convenience:

```
typedef  int  (*DPSEventFilterFunc)(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	Turn event coalescing on or off
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <dpsclient/dpsclient.h>

int **DPSSetTracking**(int *flag*)

DESCRIPTION

DPSSetTracking() turns event coalescing on or off for the current context. If *flag* is 0, event 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 count down for wait cursor
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <dpsclient/dpsclient.h>

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()**, **setwaitcursorenabled**

DPSTraceContext()

SUMMARY Control debugging tracing of context's input and output

LIBRARY libNeXT_s.a

SYNOPSIS

#import <dpsclient/dpsclient.h>

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. When 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 the UNIX® standard error file, **stderr**. This copy can be useful for program debugging and optimization.

The first argument, *context*, specifies the context to be traced. An application's single context can be returned with **DPSGetCurrentContext()**. (See the *Client Library Reference Manual* for information on **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. If *flag* is YES, **DPSTraceContext()** chains a new context, known as the child context, to *context*, the parent context. (See "Chained Contexts" in the "Application Support" section of the *Client Library Reference Manual.*) The new context receives an ASCII version of the PostScript code that's sent to the parent context. It also receives a copy of any values returned from the parent context to the client process. In the tracing output, values returned to the application are marked by the prepended string:

```
% value returned ==>
```

If *flag* is NO, the child context is unchained and destroyed.

For applications based on the Application Kit, there are two preferable methods for turning on 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.

DPSTraceEvents()

SUMMARY	Control debugging tracing of a context's events
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <dpsclient/dpsclient.h>

void **DPSTraceEvents**(DPSContext *context*, int *flag*)

DESCRIPTION

DPSTraceEvents() controls the tracing of events. When event tracing is enabled, information about each event that the application receives is sent to the UNIX standard error file, **stderr**. This information can be useful for program debugging and optimization.

The first argument, *context*, specifies the context to be traced. An application's single context can be returned with **DPSGetCurrentContext**(). (See the *Client Library Reference Manual* for information on **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 event tracing is enabled. If *flag* is YES, event tracing is enabled; if *flag* is NO, it's disabled.

When tracing is enabled and the application receives an event, the event record's components are listed. 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. (See **dpsclient/event.h** for the structure of the event record.) The format 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. The following table lists the contents of the data field according to event type.

Event Type	**Data**
NX_LMOUSEDOWN NX_LMOUSEUP NX_RMOUSEDOWN NX_RMOUSEUP	**data.mouse.eventNum**, **data.mouse.click**
NX_KEYDOWN NX_KEYUP	**data.key.repeat**, **data.key.charSet**, **data.key.charCode**, **data.key.keyCode**, **data.key.keyData**
NX_MOUSEENTERED NX_MOUSEEXITED	**data.tracking.eventNum**, **data.tracking.trackingNum**, **data.tracking.userData**
NX_MOUSEMOVED NX_LMOUSEDRAGGED NX_RMOUSEDRAGGED NX_FLAGSCHANGED And all other event types	**data.compound.subtype**, **data.compound.misc.L[0]**, **data.compound.misc.L[1]**

For applications based on the Application Kit, there are two more convenient methods for turning on 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.

DPSUndefineUserObject() → See DPSDefineUserObject()

NXAllocErrorData(), NXResetErrorData()

SUMMARY Manage the error data buffer

LIBRARY libNeXT_s.a

SYNOPSIS

#import <objc/error.h>

void **NXAllocErrorData**(int *size*, void ***data*)
void **NXResetErrorData**(void)

DESCRIPTION

These functions handle the error buffer, which is used to pass error data to an error handler. When an error occurs, **NX_RAISE()** is called with two arguments that point to an arbitrary amount of data about the error. If an error handler can't respond to the error, the error code and associated data are passed to the next higher-level handler.

NXAllocErrorData() allocates *size* amount of space in the error buffer, increasing the size of the buffer if necessary. The *data* argument points to a pointer to the data in the buffer. To empty and free the buffer, call **NXResetErrorData()**. If you're using the Application Kit, the buffer is freed for you upon each pass through the event loop.

SEE ALSO

NX_RAISE(), **NXDefaultTopLevelErrorHandler()**

NXAlphaComponent() → See NXRedComponent()

NXAtEOS() → See NXSeek()

NXAttachPopUpList(), NXCreatePopUpListButton()

SUMMARY Set up a pop-up list

LIBRARY libNeXT_s.a

SYNOPSIS

#import <appkit/appkit.h>

void **NXAttachPopUpList**(id *button*, PopUpList *popUpList*)
id **NXCreatePopUpListButton**(PopUpList *popUpList*)

DESCRIPTION

These functions make it easy to use the PopUpList class, which is described in more detail in Chapter 3. **NXCreatePopUpListButton()** returns a new Button object that will activate the pop-up list specified by *popUpList*.

NXAttachPopUpList() modifies *button* so that it activates *popUpList*. In addition, if *button* already has a target and an action, then they are used whenever a selection is made from the pop-up list.

RETURN

NXCreatePopUpListButton() returns a new Button object.

NXBeep()

SUMMARY	Play the system beep
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <appkit/publicWraps.h>

void **NXBeep**(void)

DESCRIPTION

This function plays the system beep. Users can select a sound to be played as the system beep through the Preferences application.

NXBeginTimer(), NXEndTimer()

SUMMARY	Set up timer events
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <appkit/timer.h>

NXTrackingTimer ***NXBeginTimer(**NXTrackingTimer **timer*, double *delay*,
double *period*)
void **NXEndTimer**(NXTrackingTimer **timer*)

DESCRIPTION

These functions start up and end a timed entry that puts timer events in the event queue at specified intervals. They ensure that the modal event loop will get a stream of events even if none are being generated by the Window Server.

NXBeginTimer()'s *delay* argument specifies the number of seconds after which timer events will begin to be added to the event queue; an event will then be added every *period* seconds. The first argument, *timer*, is a pointer to an NXTrackingTimer structure, which is defined in the header file **appkit/timer.h**. You don't have to initialize this argument. If you pass a NULL pointer, memory will be allocated for the structure. Since timer events are usually needed only within a modal event loop, it's generally better to declare the structure as a local variable on the stack.

NXEndTimer() stops the flow of timer events. Its argument should be a pointer to the NXTrackingTimer structure used by **NXBeginTimer**(). If memory had been allocated for the structure, **NXEndTimer**() frees it.

RETURN

NXBeginTimer() returns a pointer to the NXTrackingTimer structure it uses.

NXBlackComponent() → See NXRedComponent()

NXBlueComponent() → See NXRedComponent()

NXBPSFromDepth() → See NXColorSpaceFromDepth()

NXBrightnessComponent() → See NXRedComponent()

NXChangeAlphaComponent() → See NXChangeRedComponent()

NXChangeBlackComponent() → See NXChangeRedComponent()

NXChangeBlueComponent() → See NXChangeRedComponent()

NXChangeBrightnessComponent() → See NXChangeRedComponent()

NXChangeBuffer() → See NXStreamCreate()

NXChangeCyanComponent() → See NXChangeRedComponent()

NXChangeGrayComponent() → See NXChangeRedComponent()

NXChangeGreenComponent() → See NXChangeRedComponent()

NXChangeHueComponent() → See NXChangeRedComponent()

NXChangeMagentaComponent() → See NXChangeRedComponent()

NXChangeRedComponent(), NXChangeGreenComponent(), NXChangeBlueComponent(), NXChangeCyanComponent(), NXChangeMagentaComponent(), NXChangeYellowComponent(), NXChangeBlackComponent(), NXChangeHueComponent(), NXChangeSaturationComponent(), NXChangeBrightnessComponent(), NXChangeGrayComponent(), NXChangeAlphaComponent()

SUMMARY	Modify a color by changing one of its components
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <appkit/color.h>

NXColor **NXChangeRedComponent(**NXColor *color*, float *red***)**
NXColor **NXChangeGreenComponent(**NXColor *color*, float *green***)**
NXColor **NXChangeBlueComponent(**NXColor *color*, float *blue***)**
NXColor **NXChangeCyanComponent(**NXColor *color*, float *cyan***)**
NXColor **NXChangeMagentaComponent(**NXColor *color*, float *magenta***)**
NXColor **NXChangeYellowComponent(**NXColor *color*, float *yellow***)**
NXColor **NXChangeBlackComponent(**NXColor *color*, float *black***)**
NXColor **NXChangeHueComponent(**NXColor *color*, float *hue***)**
NXColor **NXChangeSaturationComponent(**NXColor *color*, float *saturation***)**
NXColor **NXChangeBrightnessComponent(**NXColor *color*, float *brightness***)**
NXColor **NXChangeGrayComponent(**NXColor *color*, float *gray***)**
NXColor **NXChangeAlphaComponent(**NXColor *color*, float *alpha***)**

DESCRIPTION

These functions alter one component of a color value and return the new color. The first argument, *color*, is the color to be altered and the second argument is the new value for the altered component. For example, the code below specifies a color with a greater red content than the standard brown:

```
NXColor redBrown = NXChangeRedComponent(NX_COLORBROWN, 0.9);
```

Note that the *color* argument is used as a reference for creating a new color value; it is not itself changed.

Values passed for the altered component should lie between 0.0 and 1.0; out-of-range values will be lowered to 1.0 or raised to 0.0. NX_NOALPHA can be passed to **NXChangeAlphaComponent()** to remove any specification of coverage from the color.

RETURN

These functions return an NXColor structure that, except for the altered component, represents a color identical to the one passed as an argument.

SEE ALSO

NXRedComponent(), **NXSetColor()**, **NXConvertRGBAToColor()**, **NXConvertColorToRGBA()**, **NXEqualColor()**, **NXReadColor()**

NXChangeSaturationComponent() → See NXChangeRedComponent()

NXChangeYellowComponent() → See NXChangeRedComponent()

NXChunkCopy() → See NXChunkMalloc()

NXChunkGrow() → See NXChunkMalloc()

NXChunkMalloc(), NXChunkRealloc(), NXChunkGrow(), NXChunkCopy(), NXChunkZoneMalloc(), NXChunkZoneRealloc(), NXChunkZoneGrow(), NXChunkZoneCopy()

SUMMARY	Manage variable-sized arrays of records
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <appkit/chunk.h>

NXChunk ***NXChunkMalloc**(int *growBy*, int *initUsed*)
NXChunk ***NXChunkRealloc**(NXChunk **pc*)
NXChunk ***NXChunkGrow**(NXChunk **pc*, int *newUsed*)
NXChunk ***NXChunkCopy**(NXChunk **pc*, NXChunk **dpc*)
NXChunk ***NXChunkZoneMalloc**(int *growBy*, int *initUsed*, NXZone **zone*)
NXChunk ***NXChunkZoneRealloc**(NXChunk **pc*, NXZone **zone*)
NXChunk ***NXChunkZoneGrow**(NXChunk **pc*, int *newUsed*, NXZone **zone*)
NXChunk ***NXChunkZoneCopy**(NXChunk **pc*, NXChunk **dpc*, NXZone **zone*)

DESCRIPTION

A Text object uses these functions to manage variable-sized arrays of records. For general storage management, use objects of the Storage or List class.

These functions are paired (for example, **NXChunkZoneMalloc()** and **NXChunkMalloc()**): One function lets you specify a zone and one doesn't. Those functions that don't take a zone argument operate within the default zone, as returned by **NXDefaultMallocZone()**. In all other respects, the two types of functions are identical. In the following discussion, statements concerning one member of a function pair apply equally well to the other member.

Arrays that are managed by these functions must have as their first element an NXChunk structure, as defined in **appkit/chunk.h**:

```
typedef struct _NXChunk {
    short  growby;      /* Increment to grow by */
    int    allocated;   /* Number of bytes allocated */
    int    used;        /* Number of bytes used */
} NXChunk;
```

For example, assuming an **account** structure has been declared, an **accountArray** structure is declared as:

```
typedef struct _accountArray {
    NXChunk  chunk;
    account  record[1];
} accountArray;
```

The NXChunk structure stores three values: **growby** specifies how many additional bytes of storage will be allocated when **NXChunkRealloc()** is called; **allocated** stores the number of bytes currently allocated for the array; and **used** stores the number of bytes currently used by the array's elements.

Note: The values recorded in the NXChunk element don't take into account the size of the NXChunk element itself. However, the functions described here preserve space for this element. You don't need to take into account the size of the array's NXChunk when using these functions.

Use **NXChunkMalloc()** to initially allocate memory for the array. The amount of memory allocated is equal to *initUsed*. If *initUsed* is 0, *growby* bytes is allocated. The array's NXChunk element records the value of *growby* and the amount of memory allocated for the array.

NXChunkRealloc() increases the amount of memory available for the array identified by the pointer *pc*. The amount of memory allocated depends on the value of the **growby** member of the array's NXChunk element. If the value is 0, the space for elements is doubled; otherwise the array's size increases by **growby** bytes. The **allocated** member of the array's NXChunk element stores the new size of the array.

NXChunkGrow() increases the size of the array identified by the pointer *pc* by a specific amount. The *newUsed* argument specifies the array's new size in bytes. If the **growby** member of the array's NXChunk element is 0, the array grows to the size specified by *newUsed*. Otherwise, the array grows to the larger of **growby** and *newUsed*. In either case, the size of the array changes only if the new size is larger than the old one.

NXChunkCopy() copies the array identified by the pointer *pc* to the array identified by the pointer *dpc* and returns a pointer to the copy. Since the new array may be relocated in memory, the returned pointer may be different than *dpc*.

RETURN

Each function returns a pointer to an array's NXChunk element. **NXChunkMalloc()** returns a pointer to the newly allocated array, **NXChunkRealloc()** and **NXChunkGrow**() return pointers to the resized arrays, and **NXChunkCopy**() returns a pointer to the copy of the array.

NXChunkRealloc() → See NXChunkMalloc()

NXChunkZoneCopy() → See NXChunkMalloc()

NXChunkZoneGrow() → See NXChunkMalloc()

NXChunkZoneMalloc() → See NXChunkMalloc()

NXChunkZoneRealloc() → See NXChunkMalloc()

NXClose()

SUMMARY	Close a stream
LIBRARY	libsys_s.a

SYNOPSIS

#import <streams/streams.h>

void **NXClose**(NXStream **stream*)

DESCRIPTION

This function closes the stream given as its argument. If the stream had been opened for writing, it's flushed first. (The NXStream structure is defined in the header file **stream/streams.h**.)

If the stream had been a file stream, the storage used by the stream is freed, but the file descriptor isn't closed. See the UNIX manual page on **close()** for information about closing a file descriptor. If the stream had been on memory, the internal buffer is truncated to the size of the data in it. (Calling **NXClose()** on a memory stream is equivalent to **NXCloseMemory**() with the constant NX_TRUNCATEBUFFER.)

EXCEPTIONS

NXClose() raises an NX_illegalStream exception if the stream passed in is invalid.

SEE ALSO

NXCloseMemory()

NXCloseMemory() → See NXOpenMemory()

NXCloseTypedStream() → See NXOpenTypedStream()

NXColorSpaceFromDepth(), NXBPSFromDepth(), NXNumberOfColorComponents(), NXGetBestDepth()

SUMMARY	Get information about color space and window depth
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <appkit/graphics.h>

NXColorSpace **NXColorSpaceFromDepth**(NXWindowDepth *depth*)
int **NXBPSFromDepth**(NXWindowDepth *depth*)
int **NXNumberOfColorComponents**(NXColorSpace *space*)
BOOL **NXGetBestDepth**(NXWindowDepth **depth*, int *numColors*, int *bps*)

DESCRIPTION

The first of these functions, **NXColorSpaceFromDepth()**, maps an enumerated value for window depth into the corresponding enumerated value for color space. The *depth* argument can be any of the following:

NX_TwoBitGrayDepth
NX_EightBitGrayDepth
NX_TwelveBitRGBDepth
NX_TwentyFourBitRGBDepth

The value returned will be one of the NXColorSpace values in this list:

NX_OneIsBlackColorSpace
NX_OneIsWhiteColorSpace
NX_RGBColorSpace
NX_CMYKColorSpace

NX_TwoBitGrayDepth and NX_EightBitGrayDepth map to NX_OneIsWhiteColorSpace.

The second function, **NXBPSFromDepth()**, extracts the number of bits per sample (bits per pixel in each color component) from a window *depth.*

The third function, **NXNumberOfColorComponents()**, similarly extracts the number of color components from a color *space.* The value returned will be 1, 3, or 4.

The fourth function, **NXGetBestDepth()**, finds the best window depth for an image with a given number of color components, *numColors*, and a given bits per sample, *bps*. The depth is returned by reference in the variable specified by *depth.* It will be one of the enumerated values listed above. If the depth provided exactly matches the requirements of *numColors* and *bps*, or is deeper than required, this function returns YES. If the depth isn't deep enough for *numColors* and *bps*, but is the best available, it returns NO.

RETURN

NXColorSpaceFromDepth() returns the color space that matches a given window *depth.* **NXBPSFromDepth()** returns the number of bits per sample for a given window *depth.* **NXNumberOfColorComponents()** returns the number of color components in a given color *space.* **NXGetBestDepth()** returns YES if it can provide a window depth deep enough for *numColors* and *bps*, and NO if it can't.

NXCompareHashTables() → See NXCreateHashTable()

NXCompleteFilename()

SUMMARY Match an incomplete filename

LIBRARY libNeXT_s.a

SYNOPSIS

#import <appkit/SavePanel.h>

int **NXCompleteFilename**(char **path*, int *maxPathSize*);

DESCRIPTION

NXCompleteFilename is used by the SavePanel class to determine the number of files matching an incomplete pathname. *path* is a pointer to a buffer containing an incomplete pathname. *maxPathSize* is the size of the buffer, *not* the length of *path* as determined by **strlen**(*path*).

RETURNS

This function returns the number of files that match the incomplete name. By reference, *path* returns up to *maxPathSize* characters of the path to the first file matching the incomplete name.

NXContainsRect() → See NXMouseInRect()

NXConvertCMYKAToColor() → See NXConvertRGBAToColor()

NXConvertCMYKToColor() → See NXConvertRGBAToColor()

NXConvertColorToCMYK() → See NXColorToRGBA()

NXConvertColorToCMYKA() → See NXColorToRGBA()

NXConvertColorToGray() → See NXColorToRGBA()

NXConvertColorToGrayAlpha() → See NXColorToRGBA()

NXConvertColorToHSB() → See NXColorToRGBA()

NXConvertColorToHSBA() → See NXColorToRGBA()

NXConvertColorToRGB() → See NXColorToRGBA()

NXConvertColorToRGBA(), NXConvertColorToCMYKA(), NXConvertColorToHSBA(), NXConvertColorToGrayAlpha(), NXConvertColorToRGB(), NXConvertColorToCMYK(), NXConvertColorToHSB(), NXConvertColorToGray()

SUMMARY Convert a color value to its standard components

LIBRARY libNeXT_s.a

SYNOPSIS

#import <appkit/color.h>

void **NXConvertColorToRGBA**(NXColor *color*, float **red*, float **green*, float **blue*, float **alpha*)
void **NXConvertColorToCMYKA**(NXColor *color*, float **cyan*, float **magenta*, float **yellow*, float **black*, float **alpha*)
void **NXConvertColorToHSBA**(NXColor *color*, float **hue*, float **saturation*, float **brightness*, float **alpha*)
void **NXConvertColorToGrayAlpha**(NXColor *color*, float **gray*, float **alpha*)
void **NXConvertColorToRGB**(NXColor *color*, float **red*, float **green*, float **blue*)
void **NXConvertColorToCMYK**(NXColor *color*, float **cyan*, float **magenta*, float **yellow*, float **black*)
void **NXConvertColorToHSB**(NXColor *color*, float **hue*, float **saturation*, float **brightness*)
void **NXConvertColorToGray**(NXColor *color*, float **gray*)

DESCRIPTION

These functions convert a color value, *color*, to its standard components. The first argument to each function is the NXColor data structure to be converted. Subsequent arguments point to **float** variables where the component values can be returned by reference.

The conversion can be to any set of components that might be used to specify a color value:

- Red, green, and blue (RGB) components
- Cyan, magenta, yellow, and black (CMYK) components
- Hue, saturation, and brightness (HSB) components
- A single component for gray scale images

A color initially specified by one set of components can be converted to another set. For example:

```
NXColor  color;
float    hue, saturation, brightness;

color = NXConvertRGBToColor(0.8, 0.3, 0.15);
NXConvertColorToHSB(color, &hue, &saturation, &brightness);
```

The first four functions in the list above report the coverage component, *alpha*, included in the color value, as well as the color components. The second four report only the color components; they're macros and are defined on the corresponding functions, but ignore the *alpha* argument.

The **float** values returned by reference will lie in the range 0.0 through 1.0. The value returned for the coverage component will be NX_NOALPHA if *color* doesn't include a coverage specification.

SEE ALSO

NXConvertRGBAToColor(), **NXSetColor()**, **NXEqualColor()**, **NXRedComponent()**, **NXChangeRedComponent()**, **NXReadColor()**

NXConvertGlobalToWinNum() → See NXConvertWinNumToGlobal ()

NXConvertGrayAlphaToColor() → See NXConvertRGBAToColor()

NXConvertGrayToColor() → See NXConvertRGBAToColor()

NXConvertHSBAToColor() → See NXConvertRGBAToColor()

NXConvertHSBToColor() → See NXConvertRGBAToColor()

NXConvertRGBAToColor(), NXConvertCMYKAToColor(), NXConvertHSBAToColor(), NXConvertGrayAlphaToColor(), NXConvertRGBToColor(), NXConvertCMYKToColor(), NXConvertHSBToColor(), NXConvertGrayToColor()

SUMMARY	Specify a color value
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <appkit/color.h>

NXColor **NXConvertRGBAToColor**(float *red*, float *green*, float *blue*, float *alpha*)
NXColor **NXConvertCMYKAToColor**(float *cyan*, float *magenta*, float *yellow*, float *black*, float *alpha*)
NXColor **NXConvertHSBAToColor**(float *hue*, float *saturation*, float *brightness*, float *alpha*)
NXColor **NXConvertGrayAlphaToColor**(float *gray*, float *alpha*)
NXColor **NXConvertRGBToColor**(float *red*, float *green*, float *blue*)
NXColor **NXConvertCMYKToColor**(float *cyan*, float *magenta*, float *yellow*, float *black*)
NXColor **NXConvertHSBToColor**(float *hue*, float *saturation*, float *brightness*)
NXColor **NXConvertGrayToColor**(float *gray*)

DESCRIPTION

These functions specify a color by its standard components and return an NXColor structure for the color. In the Application Kit, a color can be specified in any of four ways:

- By its red, green, and blue components (RGB)
- By its cyan, magenta, yellow, and black components (CMYK)
- By its hue, saturation, and brightness components (HSB)
- On a gray scale

No matter how they're specified, all color values are stored as the NXColor data type. The internal format of this type is unspecified; it should be set only through these functions or as one of the constants defined for pure colors, such as NX_COLORORANGE or NX_COLORWHITE.

The NXColor structure includes provision for a coverage component, *alpha*, which can be specified at the same time as the color. The first four functions listed above specify both color and coverage. The last four specify only color; they're defined as macros that work through the corresponding functions by passing NX_NOALPHA for the *alpha* argument.

Except for NX_NOALPHA, all values passed for color and coverage components should lie in the range 0.0 through 1.0; higher values will be reduced to 1.0 and lower ones raised to 0.0.

RETURN

Each of these functions and macros returns an NXColor structure for the color specified.

SEE ALSO

NXConvertColorToRGBA(), **NXSetColor**(), **NXEqualColor**(), **NXRedComponent**(), **NXChangeRedComponent**(), **NXReadColor**()

NXConvertRGBToColor() → See NXConvertRGBAToColor()

NXConvertWinNumToGlobal(), NXConvertGlobalToWinNum ()

SUMMARY Convert local and global window numbers

LIBRARY libNeXT_s.a

SYNOPSIS

#import <appkit/ publicWraps.h>

void **NXConvertWinNumToGlobal**(int *winNum*, unsigned int **globalNum*)
void **NXConvertGlobalToWinNum**(int *globalNum*, unsigned int **winNum*)

DESCRIPTION

These functions allow two or more applications to refer to the same window. In the rare cases where this is necessary, the global window number, which has been automatically assigned by the Window Server, is used rather than the local window number, which is assigned by the application.

NXConvertWinNumToGlobal() 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 to access the window. To convert window numbers in the opposite direction, give the global number as an argument for **NXConvertGlobalToWinNum**(); this function places the appropriate local number in the variable specified by *winNum*.

NXCopyBits()

SUMMARY Copy an image

LIBRARY libNeXT_s.a

SYNOPSIS

#import <appkit/graphics.h>

void **NXCopyBits**(int *gstate*, NXRect **aRect*, const NXPoint **aPoint*)

DESCRIPTION

NXCopyBits() uses the **composite** operator to copy the pixels in the rectangle specified by *aRect* to the location specified by *aPoint*.

The source rectangle is defined in the graphics state designated by the *gstate* user object. If *gstate* is NXNullObject, the current graphics state is assumed. NXNullObject is declared in **appkit/Application.h**.

The *aPoint* destination is defined in the current graphics state.

SEE ALSO

composite operator

NXCopyCurrentGState() → See NXSetGState()

NXCopyHashTable() → See NXCreateHashTable()

NXCopyInputData(), NXCopyOutputData()

SUMMARY Save data received in a remote message

LIBRARY libNeXT_s.a

SYNOPSIS

#import <appkit/ Listener.h>

char ***NXCopyInputData**(int *parameter*)
char ***NXCopyOutputData**(int *parameter*)

DESCRIPTION

These functions each return a pointer to memory containing data passed from one application to another in a remote message. **NXCopyInputData**() is used for data received by a Listener object, and **NXCopyOutputData**() is used for return data received back by a Speaker.

Data received by a Listener in a remote message is guaranteed only for the duration of the receiving application's response to the message. Return data passed back to a Speaker is guaranteed only until the Speaker receives another return message. Therefore, you must copy any data you wish to keep.

If the data is passed in-line (if it's not too large to fit within the Mach message), these functions allocate memory for the data, copy it, and return a pointer to the copy. However, it's likely that more memory will be allocated than is required for the copy. Both functions use **vm_allocate**(), which provides memory in multiples of a page.

Therefore, for in-line data, it's more efficient for you to allocate the memory yourself, using **malloc**() or **NX_MALLOC**(), then copy the data using a standard library function like **strcpy**().

For out-of-line data (data that's too large to fit within the Mach message itself, so that only a pointer to it is passed), it's generally more efficient to use **NXCopyInputData**() and **NXCopyOutputData**() to save a copy. Both functions ensure that the Listener or Speaker won't free the out-of-line data. Both return a pointer to the data without actually copying it.

The memory returned by these functions should be freed using **vm_deallocate**(), rather than **free**().

The data to be saved is identified by *parameter*, an index into the list of parameters declared for the Objective-C method that sends or receives the remote message. Indices begin at 0, and byte arrays count as a single parameter even though they're declared as a combination of a pointer to the array and an integer that counts the number of bytes in the array.

The examples below illustrate how these these functions are used. In the first, a Listener receives a **translateGaelic::toWelsh::ok:** message, a fictitious message which requests the receiving application to exchange Gaelic text for the equivalent Welsh version. If the application needs to save the original text, it would copy it, using **NXCopyInputData**(), in the method it implements to respond to the message:

```
char  *originalText;

- (int)translateGaelic:(char *)gaelicText
            :(int)gaelicLength
            toWelsh:(char *)welshText
            :(int *)welshLength
            ok:(int *)flag
{
    if ( gaelicLength >= vm_page_size )
        originalText = NXCopyInputData(0);
    . . .
}
```

The application that sends a **translateGaelic::toWelsh::ok:** message would save the returned text, using **NXCopyOutputData**(), immediately after sending the remote message:

```
char  *newText;
int    newLength;
int    error, success;

error = [mySpeaker translateGaelic:someText
                    :strlen(someText)
                    toWelsh:&newText
                    :&newLength
                    ok:&success];
if ( !error && success )
    newText = NXCopyOutputData(1);
```

RETURN

Both functions return a pointer to memory containing data identified by the *parameter* index, or a NULL pointer if the data can't be provided.

NXCopyOutputData() → See NXCopyInputData()

NXCopyStringBuffer() → See NXUniqueString()

NXCopyStringBufferFromZone() → See NXUniqueString()

NXCountHashTable() → See NXHashInsert()

NXCountWindows(), NXWindowList()

SUMMARY Get information about an application's windows

LIBRARY libNeXT_s.a

SYNOPSIS

#import <appkit/publicWraps.h>

void **NXCountWindows**(int **count*)
void **NXWindowList**(int *size*, int *list*[])

DESCRIPTION

NXCountWindows() counts the number of on-screen windows belonging to the application; it returns the number by reference in the variable specified by *count*.

NXWindowList() 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 **NXCountWindows()** to specify the size of the array for **NXWindowList()**.

NXCreateChildZone() → See NXZoneMalloc()

NXCreateHashTable(), NXCreateHashTableFromZone(), NXFreeHashTable(), NXEmptyHashTable(), NXResetHashTable(), NXCopyHashTable(), NXCompareHashTables(), NXPtrHash(), NXStrHash(), NXPtrIsEqual(), NXStrIsEqual(), NXNoEffectFree(), NXReallyFree()

SUMMARY	Create and free a hash table
LIBRARY	libsys_s.a

SYNOPSIS

#import <objc/hashtable.h>

NXHashTable ***NXCreateHashTable**(NXHashTablePrototype *prototype*, unsigned *capacity*, const void **info*)
NXHashTable ***NXCreateHashTableFromZone**(NXHashTablePrototype *prototype*, unsigned *capacity*, const void **info*, NXZone **zone*)
void **NXFreeHashTable**(NXHashTable **table*)
void **NXEmptyHashTable**(NXHashTable **table*)
void **NXResetHashTable**(NXHashTable **table*)
NXHashTable ***NXCopyHashTable**(NXHashTable **table*)
BOOL **NXCompareHashTables**(NXHashTable **table1*, NXHashTable **table2*)
unsigned **NXPtrHash**(const void **info*, const void **data*)
unsigned **NXStrHash**(const void **info*, const void **data*)
int **NXPtrIsEqual**(const void **info*, const void **data1*, const void **data2*)
int **NXStrIsEqual**(const void **info*, const void **data1*, const void **data2*)
void **NXNoEffectFree**(const void **info*, void **data*)
void **NXReallyFree**(const void **info*, void **data*)

DESCRIPTION

These functions set up, copy, and free a hash table. A hash table provides an efficient means of manipulating elements of an unordered set of data. A data element is stored by computing a hash function—or hashing—on the element to be stored. The value of the hashing function, sometimes called the key, is used to determine the location at which to store the data. The functions described under **NXHashInsert**() insert, remove, and search for a data element; they also count the number of elements and iterate over all elements in a hash table.

To create a hash table, call **NXCreateHashTable**() or **NXCreateHashTableFromZone**(). These functions differ only in that the first one creates the hash table in the default zone, as returned by **NXDefaultMallocZone**(), and the second lets you specify a zone. Only **NXCreateHashTable**() will be discussed below.

The first argument to **NXCreateHashTable()** is a NXHashTablePrototype structure, which is defined in **objc/hashtable.h** and shown below. This structure requires you to specify three functions, a hashing function, a comparison function that determines whether two data elements are equal, and a freeing function that frees a given data element in the table:

```
typedef struct {
    unsigned  (*hash)(const void *info, const void *data);
    int       (*isEqual)(const void *info, const void *data1,
                         const void *data2);
    void      (*free)(const void *info, void *data);
    int         style;
} NXHashTablePrototype;
```

The hashing function must be defined such that if two data elements are equal, as defined by the comparison function, the values produced by hashing on these elements must also be equal. Also, data elements must remain invariant if the value of the hashing function depends on them; for example, if the hashing function operates directly on the characters of a string, that string can't change. The comparison function must return true if and only if the two data elements being compared are equal. The third function specifies how a data element is to be freed. The *style* field is reserved for future use; currently, it should be passed in as 0.

As shown, the third argument for **NXCreateHashTable()**, *info*, is passed as the first argument to the hashing, comparison, and freeing functions. You can use *info* to modify or add to the effects produced by these functions. For example, the comparison function can be modified to return a certain value if the elements being compared are similar to each other but not exactly equal.

For convenience, functions for hashing pointers, integers, and strings and for comparing them have already been defined; two different freeing functions are also provided. **NXPtrHash()** hashes the address bits of *data* and returns a key for storing the data. **NXPtrIsEqual()** returns nonzero if *data1* is equal to *data2* and 0 if they're not equal. These functions can be used for pointers or for data of type **int**. Similarly, **NXStrHash()** returns a key for the string passed in as *data*, and **NXStrIsEqual()** checks whether two strings are equal. **NXReallyFree()** frees the *data* element passed in, allowing its key to be reused. **NXNoEffectFree()**, as its name implies, has no effect.

The *info* argument for all six of these functions isn't used. If you want to hash data other than pointers or strings, or if you want to use the *info* argument, you need to write your own hashing, comparison, and freeing functions.

In addition to the hashing, comparison, and freeing functions, four different prototypes have been predefined. The prototype for pointers (which can also be used for data of type **int**) and the one for strings both use the functions described above:

```
const NXHashTablePrototype NXPtrPrototype = {
    NXPtrHash, NXPtrIsEqual, NXNoEffectFree, 0
};

const NXHashTablePrototype NXStrPrototype = {
    NXStrHash, NXStrIsEqual, NXNoEffectFree, 0
};
```

The following example shows how to use NXPtrPrototype to create a hash table for storing a set of pointers or data of type **int**:

```
NXHashTable  *myHashTable;
myHashTable = NXCreateHashTable(NXPtrPrototype, 0, NULL);
```

Note that you pass the NXPtrPrototype structure as an argument, not a pointer to it. **NXCreateHashTable()** returns a pointer to an NXHashTable structure, which is defined in the header file **objc/hashtable.h**.

The other two prototypes create a hash table for storing a set of structures; the first element of each structure will be used as the key. NXPtrStructKeyPrototype expects the first element to be a pointer, and NXStrStructKeyPrototype expects a string. The free function for both these prototypes is **NXReallyFree().**

NXCreateHashTable()'s second argument, *capacity*, is only a hint; you can just pass 0 to create a minimally sized table. As more space is needed, it will be automatically and efficiently allocated.

NXFreeHashTable() frees each element of the specified hash table and the table itself. **NXResetHashTable()** frees each element but doesn't deallocate the table. This is useful for retaining the table's capacity. **NXEmptyHashTable()** sets the number of elements in the table to 0 but doesn't deallocate the table or the data in it.

NXCopyHashTable() returns a pointer to a copy of the hash table passed in. **NXCompareHashTables()** returns YES if the two hash tables supplied as arguments are equal. That is, each element of *table1* is in *table2*, and the two tables are the same size.

RETURN

NXCreateHashTable(), **NXCreateHashTableFromZone**(), and **NXCopyHashTable**() return pointers to the new hash tables they create.

NXCompareHashTables() returns YES if the two hash tables supplied as arguments are equal.

NXPtrHash() returns a key for storing a pointer in a hash table; **NXStrHash**() returns a key for storing a string.

NXPtrIsEqual() and **NXStrIsEqual**() return nonzero if the two data elements passed in are equal, and 0 if they're not.

SEE ALSO

NXHashInsert()

NXCreateHashTableFromZone() → See NXCreateHashTable()

NXCreatePopUpListButton() → See NXAttachPopUpList()

NXCreateZone() → See NXZoneMalloc()

NXCyanComponent() → See NXRedComponent()

NXDefaultExceptionRaiser(), NXSetExceptionRaiser(), NXGetExceptionRaiser()

SUMMARY	Set and return an exception raiser
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <objc/error.h >

void **NXDefaultExceptionRaiser**(int *code*, const void **data1*, const void **data2*)
void **NXSetExceptionRaiser**(NXExceptionRaiser **procedure*)
NXExceptionRaiser ***NXGetExceptionRaiser**(void)

DESCRIPTION

These functions set and return the procedure that's called when exceptions are raised using **NX_RAISE()**. By default, the **NXDefaultExceptionRaiser()** will be invoked by **NX_RAISE()**; this function is also what **NXGetExceptionRaiser()** returns unless you've declared your own exception raiser by using **NXSetExceptionRaiser()**, as described below.

NXDefaultExceptionRaiser() forwards the exception condition indicated by *code* and any information about the exception pointed to by *data1* and *data2* to the next error handler. Error handlers exist in a nested hierarchy, which is created by using any number of nested NX_DURING...NX_ENDHANDLER constructs and by defining a top-level error handler.

If the error has occurred outside of the domain of any handler, **NXDefaultExceptionRaiser()** invokes an uncaught exception handling function. For more information on the Application Kit's default uncaught exception handling function or to define your own, see the description of **NXSetUncaughtExceptionHandler()**. If the uncaught exception handling function can't be found, **NXDefaultExceptionRaiser()** exits.

To override the default exception raiser, call **NXSetExceptionRaiser()** and give it a pointer to the exception raising function you want to use. This function must be of type NXExceptionRaiser (that is, the same type as **NXDefaultExceptionRaiser()**), which is defined in the header file **streams/error.h** as follows:

```
typedef void NXExceptionRaiser(int code, const void *data1,
    const void *data2);
```

In other words, the function *procedure* must take three arguments of the types shown above, and it must return **void**. Once you've called **NXSetExceptionRaiser()**, subsequent calls to **NXGetExceptionRaiser()** will return a pointer to *procedure*; also, subsequent calls to **NX_RAISE()** will invoke *procedure*.

SEE ALSO

NX_RAISE(), **NXSetUncaughtExceptionRaiser()**

NXDefaultMallocZone() → See NXZoneMalloc()

NXDefaultRead() → See NXStreamCreate()

NXDefaultStringOrderTable() → See NXOrderStrings()

NXDefaultTopLevelErrorHandler(), NXSetTopLevelErrorHandler(), NXTopLevelErrorHandler()

SUMMARY	Define an error handler
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <appkit/errors.h>

void **NXDefaultTopLevelErrorHandler**(NXHandler **errorState*)
NXTopLevelErrorHandler
    ***NXSetTopLevelErrorHandler**(NXTopLevelErrorHandler **procedure*)
NXTopLevelErrorHandler ***NXTopLevelErrorHandler**(void)

DESCRIPTION

This group of a function and two macros defines the top-level error handler. The top-level handler is called when an exception is forwarded through the nested lower-level handlers up to the top level. The hierarchy of error handlers is created by using any number of nested NX_DURING...NX_ENDHANDLER constructs.

If an application doesn't define its own top-level handler, by default it will use **NXDefaultTopLevelErrorHandler()**. This function is defined and used by the Application Kit. Its only argument is a pointer to an NXHandler structure, which is defined in the header file **streams/error.h**. This file also defines **NXDefaultTopLevelErrorHandler()** as being a global variable of type NXTopLevelErrorHandler, which is defined as follows:

```
typedef void NXTopLevelErrorHandler(NXHandler *errorState);
extern NXTopLevelErrorHandler NXDefaultTopLevelErrorHandler;
```

NXDefaultTopLevelErrorHandler() calls **NXReportError()**, which executes the procedure defined to report the error that occurred. (See the description of **NXRegisterErrorReporter()** in this chapter for details about **NXReportError()**.) If an error occurred when an application's PostScript context was created or if its PostScript connection is broken, **NXDefaultTopLevelErrorHandler()** exits.

An application can override **NXDefaultTopLevelErrorHandler()** by defining its own top-level handler. This involves passing a pointer to an error-handling procedure to the macro **NXSetTopLevelErrorHandler()**. The new error-handling procedure must be of type NXTopLevelErrorHandler, which means it must take a pointer to an NXHandler as its only argument and it must return **void**.

NXTopLevelErrorHandler() returns a pointer to the current top-level handler. After a new one has been set using **NXSetTopLevelErrorHandler()**, subsequent calls to **NXTopLevelErrorHandler()** will return a pointer to the new top-level error handler.

The two macros, **NXSetTopLevelErrorHandler()** and **NXTopLevelErrorHandler()**, are defined in the header file **appkit/errors.h**.

SEE ALSO

NX_RAISE(), **NXDefaultExceptionRaiser()**, **NXRegisterErrorReporter()**

NXDefaultWrite() → See NXStreamCreate()

NXDestroyZone() → See NXZoneMalloc()

NXDivideRect() → See NXSetRect()

NXDrawALine() → See NXScanALine()

NXDrawButton(), NXDrawGrayBezel(), NXDrawGroove(), NXDrawWhiteBezel(), NXDrawTiledRects(), NXFrameRect(), NXFrameRectWithWidth()

SUMMARY	Draw a bordered rectangle
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <appkit/graphics.h>

void **NXDrawButton**(const NXRect **aRect*, const NXRect **clipRect*)
void **NXDrawGrayBezel**(const NXRect **aRect*, const NXRect **clipRect*)
void **NXDrawGroove**(const NXRect **aRect*, const NXRect **clipRect*)
void **NXDrawWhiteBezel**(const NXRect **aRect*, const NXRect **clipRect*)
NXRect ***NXDrawTiledRects**(NXRect **aRect*, const NXRect **clipRect*,
    const int **sides*, const float **grays*, int *count*)
void **NXFrameRect**(const NXRect **aRect*)
void **NXFrameRectWithWidth**(const NXRect **aRect*, NXCoord *frameWidth*)

DESCRIPTION

These functions draw rectangles with borders. **NXDrawButton()** draws the rectangle used to signify a button on a NeXT computer, **NXDrawTiledRects()** 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 3-1.

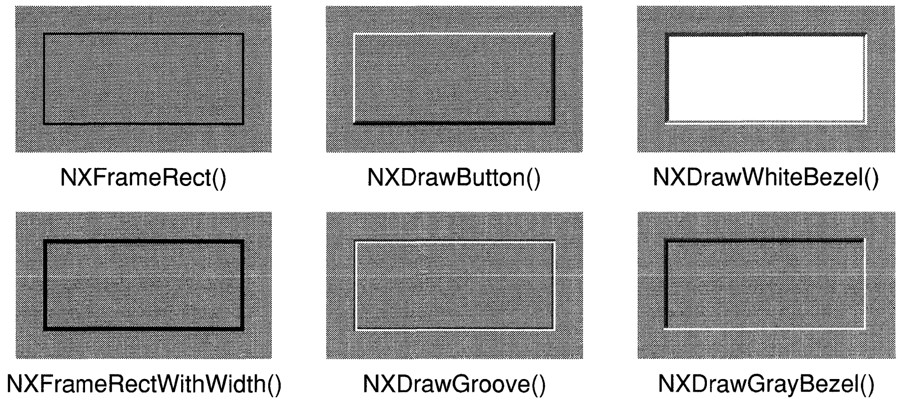


Figure 3-1. Rectangle Borders

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, **NXDrawWhiteBezel**() fills in its rectangle with white; **NXDrawButton**(), **NXDrawGrayBezel**(), and **NXDrawGroove**() 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.

NXFrameRect() and **NXFrameRectWithWidth**() draw a frame around the inside of a rectangle in the current color. **NXFrameRect**() draws a frame with a width equal to 1.0 in the current coordinate system; **NXFrameRectWithWidth**() 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, **NXDrawTiledRects**() takes three more arguments, which determine how thick the border is and what gray levels are used to form it. **NXDrawTiledRects**() works through **NXDivideRect**() 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*. **NXDrawTiledRects**() makes and draws these slices *count* number of times. **NXDivideRect**() 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, **NXDrawTiledRects**() 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,
                          NX_YMIN, NX_XMAX};
float    myGrays[] = {NX_BLACK, NX_BLACK, NX_WHITE, NX_WHITE,
                          NX_DKGRAY, NX_DKGRAY};
NXRect  *aRect;

NXDrawTiledRects(aRect, (NXRect *)0, mySides, myGrays, 6);
PSsetgray(NX_LTGRAY);
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. The constants shown in **mySides** are described in more detail in the description of **NXDivideRect**(). **myGrays** is an array that specifies the successive gray levels to be used in drawing parts of the border.

RETURN

NXDrawTiledRects() returns a pointer to the rectangle that lies within the border.

SEE ALSO

NXDivideRect()

NXDrawGrayBezel() → See NXDrawButton()

NXDrawGroove() → See NXDrawButton()

NXDrawTiledRects() → See NXDrawButton()

NXDrawWhiteBezel() → See NXDrawButton()

NXEditorFilter() → See NXFieldFilter()

NXEmptyHashTable() → See NXCreateHashTable()

NXEmptyRect() → See NXMouseInRect()

NXEndOfTypedStream()

SUMMARY	Determine whether there's more data to be read
LIBRARY	libsys_s.a

SYNOPSIS

#import <objc/typedstream.h>

BOOL **NXEndOfTypedStream**(NXTypedStream **typedStream*)

DESCRIPTION

This macro indicates whether more data is available to be read from the typed stream passed in as an argument. It should be called only on a typed stream opened for reading. (The NXTypedStream type is declared in the header file **objc/typedstream.h**. The structure itself is private since you never need to access its members.)

RETURN

NXEndOfTypedStream() returns TRUE if more data is available to be read and FALSE otherwise.

EXCEPTIONS

NXEndOfTypedStream() raises a TYPEDSTREAM_CALLER_ERROR with the message "expecting a reading stream" if the stream passed in wasn't opened for reading.

SEE ALSO

NXOpenTypedStream()

NXEndTimer() → See NXBeginTimer()

NXEqualColor()

SUMMARY	Test whether two colors are the same
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <appkit/color.h>

BOOL **NXEqualColor**(NXColor *oneColor*, NXColor *anotherColor*)

DESCRIPTION

This function compares *oneColor* to *anotherColor* and returns YES if they are, in fact, the same color. Two colors can be the same, yet be represented differently within the NXColor structure. Therefore, NXColor structures should be compared only through this function, never directly.

The coverage components of the NXColor structures are included in the comparison.

RETURN

This function returns YES if the two colors are visually identical, and NO if they're not.

SEE ALSO

NXSetColor(), **NXConvertRGBAToColor()**, **NXConvertColorToRGBA()**, **NXRedComponent()**, **NXChangeRedComponent()**, **NXReadColor()**

NXEqualRect() → See NXMouseInRect()

NXEraseRect() → See NXRectClip()

NXFieldFilter(), NXEditorFilter()

SUMMARY Filter characters entered into Text object

LIBRARY libNeXT_s.a

SYNOPSIS

#import <appkit/Text.h>

unsigned short **NXFieldFilter**(unsigned short *theChar*, int *flags*,
    unsigned short *charSet*)
unsigned short **NXEditorFilter**(unsigned short *theChar*, int *flags*,
    unsigned short *charSet*)

DESCRIPTION

These functions check each character the user types into a Text object's text. Use **NXFieldFilter**(), the Text object's default character filter, when you want the user to be able to move the selection from text field to field by pressing Return, Tab, or Shift-Tab. Use **NXEditorFilter**() when you don't want Return, Tab, and Shift-Tab interpreted in this way.

NXFieldFilter() passes on values generated by alphanumeric keys directly to the Text object for display. Values generated by Return, Tab, Shift-Tab, and the arrow keys are remapped to constants that have a special meaning for the Text object. The Text object interprets any of these constants as a movement command, a command to end the Text object's status as first responder. Based on the key pressed, the Text object's delegate can control which other object should become the first responder. **NXFieldFilter**() remaps to 0 all other values less than 0x20 and any values generated in conjunction with the Command key.

NXEditorFilter() is identical to **NXFieldFilter**() except that it passes on values corresponding to Return, Tab, and Shift-Tab directly to the Text object.

RETURN

NXFieldFilter() returns 0 (NX_ILLEGAL), the ASCII value of the character typed, or a constant the Text object interprets as a movement command. The constants are:

NX_RETURN
NX_TAB
NX_BACKTAB
NX_LEFT
NX_RIGHT
NX_UP
NX_DOWN

This function also returns 0 if a key is pressed while a Command key is held down.

NXEditorFilter()'s return values are identical to those of **NXFieldFilter()**, except that it also returns the values generated by Return, Tab, and Shift-Tab without first remapping them.

NXFilePathSearch()

SUMMARY Search for and read a file

LIBRARY libNeXT_s.a

SYNOPSIS

#import <appkit/defaults.h>
#import <defaults.h>

int **NXFilePathSearch**(const char **envVarName*, const char **defaultPath*,
 int *leftToRight*, const char **fileName*, int (**funcPtr*)(), void **funcArg*)

DESCRIPTION

NXFilePathSearch() searches a colon-separated list of directories for one or more files named *fileName*. The directory list is obtained from the environmental variable, *envVarName*, if it's available. If not, *defaultPath* is used. If *leftToRight* is true, the list of directories is searched from left to right; otherwise, it's searched right to left.

In each directory, if the file *fileName* can be accessed, the function specified by *funcPtr* is called. The function is passed two arguments, the path to the file and *funcArg*, which can contain arbitrary data for the function to use.

RETURN

If the function specified by *funcPtr* is called and returns 0 or a negative value, **NXFilePathSearch()** returns the same value. If the function returns a positive value, **NXFilePathSearch()** continues searching through the directory list for other occurrences of *fileName*. If it searches through the entire directory list, it returns 0. If it can't find a list of directories to search, it returns –1.

NXFill() → See NXStreamCreate()

NXFindPaperSize()

SUMMARY Find dimensions of specified paper type

LIBRARY libNeXT_s.a

SYNOPSIS

#import <appkit/PageLayout.h>

const NXSize ***NXFindPaperSize**(const char **paperName*)

DESCRIPTION

NXFindPaperSize() returns a pointer to an NXSize structure containing the dimensions of a sheet of paper of type *paperName*. The dimensions are given in points (72 per inch). The NXSize structure is defined in the header file **dpsclient/event.h** as follows:

```
typedef struct _NXSize {
    NXCoord  width;
    NXCoord  height;
} NXSize;
```

paperName is a character string that corresponds to one of the standard paper types used by conforming PostScript documents. For example, it could be "Letter", "Legal", or "A4". By providing the precise size of these types, this function helps programs adjust the on-screen display to the page size of the document being displayed.

RETURN

This function returns an NXSize pointer.

NXFlush()

SUMMARY Flush a stream

LIBRARY libsys_s.a

SYNOPSIS

#import <streams/streams.h>

int **NXFlush**(NXStream **stream*)

DESCRIPTION

This function flushes the buffer associated with the stream passed in as an argument. **NXFlush()** is called by **NXClose()**, so you don't have to flush the buffer before closing a stream with **NXClose()**. In some cases, you might not want to close the stream but you might want to ensure that data is actually written to the stream's destination rather than remaining in the buffer.

RETURN

NXFlush() returns the number of characters flushed from the buffer and written to the stream.

EXCEPTIONS

This function raises an NX_illegalStream exception if the stream passed in is invalid. In addition, it raises an NX_illegalWrite exception if an error occurs while flushing the stream.

NXFlushTypedStream()

SUMMARY Flush a typed stream

LIBRARY libsys_s.a

SYNOPSIS

#import <objc/typedstream.h>

void **NXFlushTypedStream**(NXTypedStream **TypedStream*)

DESCRIPTION

This function flushes the buffer associated with the typed stream passed in as an argument. **NXFlushTypedStream()** is called by **NXCloseTypedStream()**, so you don't have to flush the buffer before closing a typed stream. (The NXTypedStream type is declared in the header file **objc/typedstream.h**. The structure itself is private since you never need to access its members.)

EXCEPTIONS

NXFlushTypedStream() raises a TYPEDSTREAM_CALLER_ERROR with the message "expecting a writing stream" if the typed stream wasn't opened for writing.

SEE ALSO

NXOpenTypedStream()

NXFrameRect() → See NXDrawButton()

NXFrameRectWithWidth() → See NXDrawButton()

NXFreeAlertPanel() → See NXRunAlertPanel()

NXFreeHashTable() → See NXCreateHashTable()

NXFreeObjectBuffer() → See NXReadObjectFromBuffer()

NXGetAlertPanel() → See NXRunAlertPanel()

NXGetBestDepth() → See NXColorSpaceFromDepth()

NXGetc() → See NXPutc()

NXGetDefaultValue() → See NXRegisterDefaults()

NXGetExceptionRaiser() → See NXDefaultExceptionRaiser()

NXGetMemoryBuffer() → See NXOpenMemory()

NXGetNamedObject(), NXGetObjectName(), NXNameObject(), NXUnnameObject()

SUMMARY	Refer to objects by name
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <appkit/Application.h>

id **NXGetNamedObject**(const char **name*, id *owner*)
const char ***NXGetObjectName**(id *theObject*)
int **NXNameObject**(const char **name*, id *theObject*, id *owner*)
int **NXUnnameObject**(const char **name*, id *owner*)

DESCRIPTION

These functions permit programs that use the Application Kit to refer to objects by name. Names are assigned with Interface Builder™ or with the **NXNameObject()** function described here. When you create an object with Interface Builder, Interface Builder assigns it a default name that you can then edit or replace with a name of your own choosing. Underscores shouldn't be used as part of a name.

To distinguish among different objects with the same name, each object can also be assigned another object as an owner; the owner can be **nil**. By default, Interface Builder assigns the Application object (NXApp) as the owner of a Window, and a View's Window as the owner of that View.

NXGetNamedObject() returns the **id** of the object having the *name* and *owner* passed as arguments, or **nil** if there is no such object. Only one object can be identified by a given combination of a name and owner. **NXGetObjectName()** takes the **id** of an object and returns that object's name.

NXNameObject() assigns an object a *name* and *owner*. An object can be assigned any number of different names and owners. However, if you attempt to assign a combination of a name and owner already used to identify another (or the same) object, the assignment fails.

NXUnnameObject() disassociates an object from the combination of a *name* and *owner*. Thereafter, **NXGetNamedObject()** won't return the object when passed the *name* and *owner* as arguments.

RETURN

NXGetNamedObject() returns an object **id**, or **nil** if no object is identified by the combination of name and owner passed as arguments.

NXGetObjectName() returns the name of an object.

NXNameObject() returns 1 if it successfully assigns a name to an object, and 0 if not.

NXUnnameObject() returns 1 if it disassociates an object from the combination of name and owner passed as arguments, and 0 if the name and owner weren't associated with an object to begin with.

NXGetObjectName() → See NXGetNamedObject()

NXGetOrPeekEvent()

SUMMARY	Access event record in event queue
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <appkit/Application.h>

NXEvent ***NXGetOrPeekEvent**(DPSContext *context*, NXEvent **anEvent*, int *mask*, double *timeout*, int *threshold*, int *peek*)

DESCRIPTION

NXGetOrPeekEvent() accesses an event record in an application's event queue and returns a pointer to it. This function combines the facilities of **DPSGetEvent()** and **DPSPeekEvent()**, but unlike these client library functions, it allows your application to be journaled. Applications based on the Application Kit should use this function (or the Application class methods such as **getNextEvent:** and **peekNextEvent:into:**) to access event records.

The first argument, *context*, represents a PostScript execution context within the Window Server. Virtually all applications have only one execution context, which can be returned using **DPSGetCurrentContext()**. (See the *Client Library Reference Manual* for information on **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 an event is found, its data is copied into the storage referred to by this pointer.

mask determines the types of events sought. The header file **dpsclient/event.h** defines these constants for general use:

Constant	Event Type
NX_KEYDOWNMASK	Key-down
NX_KEYUPMASK	Key-up
NX_FLAGSCHANGEDMASK	Flags-changed
NX_LMOUSEDOWNMASK	Mouse-down, left or only mouse button
NX_LMOUSEUPMASK	Mouse-up, left or only mouse button
NX_RMOUSEDOWNMASK	Mouse-down, right mouse button
NX_RMOUSEUPMASK	Mouse-up, right mouse button
NX_MOUSEMOVEDMASK	Mouse-moved
NX_LMOUSEDRAGGEDMASK	Mouse-dragged, left or only mouse button
NX_RMOUSEDRAGGEDMASK	Mouse-dragged, right mouse button
NX_MOUSEENTEREDMASK	Mouse-entered
NX_MOUSEEXITEDMASK	Mouse-exited
NX_TIMERMASK	Timer
NX_CURSORUPDATEMASK	Cursor-update
NX_KITDEFINEDMASK	Kit-defined
NX_SYSDEFINEDMASK	System-defined
NX_APPDEFINEDMASK	Application-defined
NX_ALLEVENTS	All event types

To check for multiple types of events, you can combine these constants using the bitwise OR operator.

If an event matching the event mask isn't available in the queue, **NXGetOrPeekEvent()** 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.

threshold is an integer in the range 0 to 31 that determines which other services may be provided during a call to **NXGetOrPeekEvent**(). 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 **NXGetOrPeekEvent**() returns.

The last argument, *peek*, specifies whether **NXGetOrPeekEvent()** removes the event from the event queue. If *peek* is 0, **NXGetOrPeekEvent()** removes the record from the queue after making its data available to the application; otherwise, it leaves the record in the queue.

RETURN

If **NXGetOrPeekEvent()** finds an event record that meets the requirements of its parameters, it returns a pointer to it. Otherwise, it returns NULL.

SEE ALSO

NXJournalMouse(), **DPSGetEvent**(), **DPSPeekEvent**(), **DPSDiscardEvent**(), **DPSAddTimedEntry**(), **DPSAddPort**(), **DPSAddFD**()

NXGetTempFilename()

SUMMARY Create a temporary file name

LIBRARY libNeXT_s.a

SYNOPSIS

#import <appkit/appkit.h>

char ***NXGetTempFilename**(char **name*, int *pos*)

DESCRIPTION

This function creates a unique file name by altering the *name* argument it is passed. **NXGetTempFilename**() replaces the six characters starting at the *pos*th position within *name* with digits it generates; it then checks whether the file name is unique. If it is, the file name is returned; if not, different digits are tried until a unique name is found. **NXGetTempFilename()** is similar to the standard C function **mktemp**(), except that it can leave suffixes intact since you specify the location of the characters that get replaced.

RETURN

NXGetTempFilename() returns a unique file name.

NXGetTIFFInfo() → See NXReadTIFF()

NXGetTypedStreamZone(), NZSetTypedStreamZone()

SUMMARY Set zones for streams

LIBRARY libsys_s.a

SYNOPSIS

#import <objc/typedstream.h>

NXZone ***NXGetTypedStreamZone**(NXTypedStream **stream*)
void **NXSetTypedStreamZone**(NXTypedStream **stream*, NXZone **zone*)

DESCRIPTION

These functions let you associate a zone with a typed stream. Zones improve application performance by optimizing locality of reference. See the description under **NXZoneMalloc**() for more on allocating and freeing zones.

If no zone is set for a typed stream, its zone is the default zone. Use these functions to associate zones with the typed streams used to unarchive objects in your application. You can, for example, use these functions to be sure that objects that interact are all unarchived in the same zone.

Use **NXSetTypedStreamZone()** to set the zone used for unarchiving objects from a typed stream. Use **NXGetTypedStreamZone()** to access the zone associated with a particular typed stream.

RETURN

NXGetTypedStreamZone() returns the zone set for *stream*.
NXSetTypedStreamZone() sets *zone* as the zone for *stream*

NXGetUncaughtExceptionHandler() → See NXSetUncaughtExceptionHandler()

NXGetWindowServerMemory()

SUMMARY	Return by reference the amount of Window Server memory being used by the current Window Server context
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <appkit/Application.h>

int **NXGetWindowServerMemory(**DPSContext *context*, int **vmUsedP*,
    int **windowBackingP*, NXStream **windowDumpStream*)

DESCRIPTION

NXGetWindowServerMemory() calculates the amount of Window Server memory being used at the moment by the given Window Server context. If NULL 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 *vmUsedP*; the amount of window backing store used by windows owned by the current context is returned in the **int** pointed to by *windowBackingP*. 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, **NXGetWindowServerMemory()** 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–NULL value is passed in for *windowDumpStream*, the information returned from the **dumpwindows** operator is echoed to the **NXStream** given. This can be useful for finding out more about which windows are using up your storage.

RETURN

Normally, **NXGetWindowServerMemory**() returns 0. If NULL is passed for context and there's no current DPS Context, returns –1.

NXGrayComponent() → See NXRedComponent()

NXGreenComponent() → See NXRedComponent()

NXHashGet() → See NXHashInsert()

NXHashInsert(), NXHashInsertIfAbsent(), NXHashMember(), NXHashGet(), NXHashRemove(), NXCountHashTable(), NXInitHashState(), NXNextHashState()

SUMMARY	Manipulate the elements of a hash table
LIBRARY	libsys_s.a

SYNOPSIS

#import <objc/hashtable.h>

void *__NXHashInsert__(NXHashTable **table*, const void **data*)
void *__NXHashInsertIfAbsent__(NXHashTable **table*, const void **data*)
int **NXHashMember**(NXHashTable **table*, const void **data*)
void *__NXHashGet__(NXHashTable **table*, const void **data*)
void *__NXHashRemove__(NXHashTable **table*, const void **data*)
unsigned **NXCountHashTable**(NXHashTable **table*)
NXHashState **NXInitHashState**(NXHashTable **table*)
int **NXNextHashState**(NXHashTable **table*, NXHashState **state*, void ***data*)

DESCRIPTION

These functions manipulate the elements of a hash table that was created using **NXCreateHashTable**(). **NXCreateHashTable**(), which is described earlier in this chapter, returns a pointer to the NXHashTable structure it creates. You pass a pointer to this structure (which is defined in the header file **objc/hashtable.h**) for each of the functions described here.

NXHashInsert() inserts *data* into the hash table specified by *table*. It checks whether *data* is already in the table by using the function referred to by the *isEqual* member of the NXHashTablePrototype; this prototype is defined when the table is created. (See the description of **NXCreateHashTable()** for more information about defining the *isEqual* function.) If *data* is already in the table, the new data is inserted anyway and a pointer to the old data is returned. If *data* isn't already in the table, it's inserted and NULL is returned.

NXHashInsertIfAbsent() inserts *data* only if it isn't already in the table and then returns a pointer to *data*. If *data* is already in the table, as determined using the function referred to by *isEqual*, a pointer to the existing data is returned.

NXHashMember() checks whether *data* is in the hash table specified by *table*. If so, it returns a nonzero value; if not, it returns 0. **NXHashGet()** returns a pointer to *data* if it's in the table; if not, it returns NULL. You can use these functions if you have a pointer to the data that might be stored in the table. You can also use them if data is stored in the table as a structure containing the key for that data and if you have that key. (In a hash table, the key determines where data is stored.) For example, suppose my hash table contains data of type MyStruct and that you have a key:

```
typedef struct {
    MyKey  key;
    . . .
    } MyStruct;

MyStruct pseudo;
pseudo.key = yourKey;
```

You can then use your key on my hash table with either function:

```
int         foundIt;
foundIt = NXHashMember(myTable, &pseudo);

MyStruct  *storedData;
storedData = NXHashGet(myTable, &pseudo);
```

NXHashRemove() removes and returns a pointer to *data* unless it can't find *data* in the table, in which case it returns NULL.

NXCountHashTable() returns the number of elements in the hash table specified by *table*.

NXInitHashState() and **NXNextHashState()** iterate through the elements of a hash table. **NXInitHashState()** returns an NXHashState structure to start the iteration process; this structure is then passed to **NXNextHashState()**, which visits each element of the hash table and finally returns 0. (NXHashState is defined in the header file **objc/hashtable.h**; you shouldn't use members of this structure as they may change in the future.) The following example counts the elements in the hash table **table**:

```
unsigned count = 0;
MyData  *data;
NXHashState state = NXInitHashState(table);

while (NXNextHashState(table, &state, &data))
    count++;
```

As it progresses through the table, **NXNextHashState()** reads each element of the table into the location specified by its third argument.

RETURN

NXHashInsert() returns NULL if the given data isn't already in the table. Otherwise, it returns a pointer to the existing data.

NXHashInsertIfAbsent() returns a pointer to the given data if it isn't already in the table. Otherwise, a pointer to the existing data is returned.

NXHashMember() returns a nonzero value if it finds the given data in the hash table specified; if not, it returns 0.

NXHashGet() returns a pointer to the given data if it's in the table; if not, it returns NULL.

NXHashRemove() returns a pointer to the data it removes unless it can't find the data, in which case it returns NULL.

NXCountHashTable() returns the number of elements in the hash table.

NXInitHashState() returns an NXHashState for use with **NXNextHashState()**.

NXNextHashState() returns 0 when it has visited every element of the hash table.

SEE ALSO

NXCreateHashTable()

NXHashInsertIfAbsent() → See NXHashInsert()

NXHashMember() → See NXHashInsert()

NXHashRemove() → See NXHashInsert()

NXHighlightRect() → See NXRectClip()

NXHomeDirectory(), NXUserName()

SUMMARY Get user's home directory and name

LIBRARY libNeXT_s.a

SYNOPSIS

#import <appkit/Application.h>

const char ***NXHomeDirectory**(void)
const char ***NXUserName**(void)

DESCRIPTION

These functions return the user's home directory and name, both of which are cached at launch time. If the user's id has changed since launch time or since the last time either of these functions was called, the values are recomputed using the standard C library function **getpwuid**(). (**getpwuid**() is described in its UNIX manual page.)

RETURN

NXHomeDirectory() returns a pointer to the full pathname of the user's home directory. **NXUserName**() returns a pointer to the user's name.

NXHueComponent() → See NXRedComponent()

NXImageBitmap(), NXReadBitmap(), NXSizeBitmap()

SUMMARY Render and read bitmap images

LIBRARY libNeXT_s.a

SYNOPSIS

#import <appkit/tiff.h>

void **NXImageBitmap**(const NXRect **rect*, int *pixelsWide*, int *pixelsHigh*, int *bps*, int *spp*, int *config*, int *mask*, const void **data1*, const void **data2*, const void **data3*, const void **data4*, const void **data5*)
void **NXReadBitmap**(const NXRect **rect*, int *pixelsWide*, int *pixelsHigh*, int *bps*, int *spp*, int *config*, int *mask*, void **data1*, void **data2*, void **data3*, void **data4*, void **data5*)
void **NXSizeBitmap**(const NXRect **rect*, int **size*, int **pixelsWide*, int **pixelsHigh*, int **bps*, int **spp*, int **config*, int **mask*)

DESCRIPTION

The first of these functions, **NXImageBitmap()**, renders an image from a bitmap, binary data that describes the pixel values for the image. The second function, **NXReadBitmap()**, reads the bitmap for a rendered image using information about the image obtained from **NXSizeBitmap()**. **NXReadBitmap()** produces data that **NXImageBitmap()** can use to recreate the image. The third function, **NXSizeBitmap()**, supplies the information required by **NXReadBitmap()**.

Bitmaps can also be rendered and read through the Application Kit's NXBitmapImageRep class.

NXImageBitmap() 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 **NXImageBitmap()** describe the bitmap data, as explained in the following paragraphs.

bps is the number of bits per sample for each pixel and *spp* is the number of samples per pixel. Multiplying these two values yields the number of bits used to specify each pixel.

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.

config refers to the way data is configured in the bitmap. It should be specified as one of two constants:

NX_PLANAR	A separate data channel is used for each sample. The function provides for up to five channels, *data1*, *data2*, *data3*, *data4*, and *data5*.
NX_MESHED	Sample values are interwoven in a single channel; all values for one pixel are specified before values for the next pixel.

Figure 3-2 illustrates these two ways of configuring data.

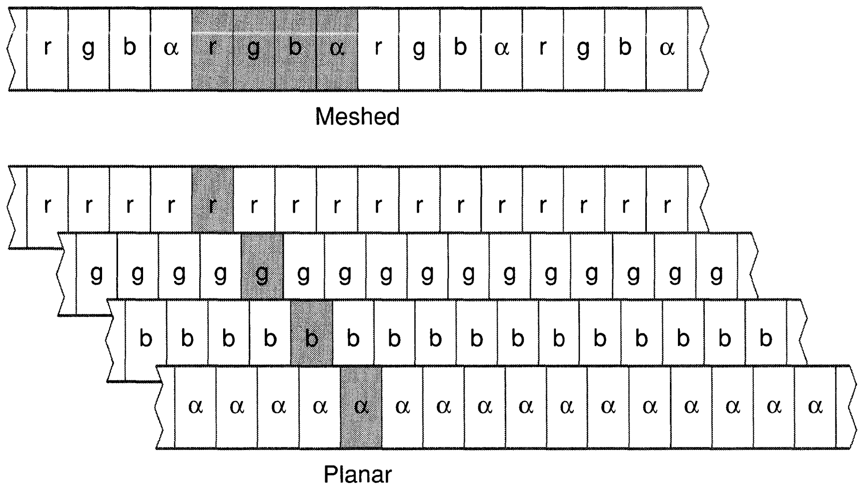


Figure 3-2. Planar and Meshed Configurations

As shown in the illustration, color samples (rgb) precede the coverage sample (α) in both configurations.

In the NeXTstep environment, gray-scale windows store pixel data in planar configuration; color windows store it in meshed configuration. **NXImageBitmap()** 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 (*bps*) and configuration (*config*) that matches the window.

mask specifies how the bitmap data is to be interpreted. It's formed by joining constants for three kinds of information (using the bitwise *OR* operator):

NX_ALPHAMASK	Coverage (alpha) values are specified. If NX_ALPHAMASK is present in *mask*, *spp* should be at least 2—one more than the number of color components.
NX_COLORMASK	Color samples are present. If NX_COLORMASK isn't included in *mask*, a gray scale is assumed.

NX_MONOTONICMASK	In a gray scale, NX_MONOTONICMASK indicates that 1 equals white and 0 equals black, as in the PostScript model. If *mask* doesn't include NX_MONOTONICMASK, the inverse scale is assumed (1 equals black, 0 equals white). NeXT computers use the PostScript gray scale. In a color system, NX_MONOTONICMASK indicates that CMYK (cyan, magenta, yellow, black) samples are specified. Its absence indicates RGB (red, green, blue) samples. This permits the function to verify that the value given for *spp* is correct. If NX_MONOTONICMASK is present in *mask*, *spp* should be 4 (5 if alpha values are also specified). If it isn't, *spp* should be 3 (4 if alpha values are also specified).

The remaining arguments, *data1* through *data5*, specify the actual bitmap data. If *config* is NX_MESHED, only *data1* is read. If *config* is NX_PLANAR, each argument should specify a separate sample.

NXReadBitmap() gets bitmap data for an existing image. It uses the PostScript **readimage** operator to read pixel values within the rectangle referred to by its first argument, *rect*. The rectangle is in the current window and is specified in the current coordinate system. If the rectangle is rotated so that its sides are no longer aligned with the screen coordinate system, **NXReadBitmap()** will read pixel values for the smallest screen-aligned rectangle enclosing the rectangle specified by *rect*.

NXReadBitmap() writes the bitmap data into the buffers specified by the *data1*, *data2*, *data3*, *data4*, and *data5* arguments. The number of actual buffers you must provide depends on whether there's a separate channel for each sample (*config*) and on the number of samples per pixel (*spp*). This information, as well as other information about the image, should be obtained directly from the device using the **NXSizeBitmap()** function.

When passed a pointer to a rectangle, **NXSizeBitmap()** gets values that **NXReadBitmap()** needs to produce a bitmap for the rectangle. It yields values that can be passed directly to **NXReadBitmap()** for the following parameters:

pixelsWide
pixelsHigh
bps
spp
config
mask

It also provides the size, in bytes, that will be required for each channel of bitmap data. **NXSizeBitmap()** works through the **currentwindowalpha** and **sizeimage** operators. The following paragraphs describe the kinds of information you could obtain from each of these operators if you were to use them directly.

If **currentwindowalpha** returns 0, the image may include some transparent paint and you'll need to obtain coverage values in addition to color values in the bitmap. Include NX_ALPHAMASK in *mask*, and make sure the alpha component is counted in *spp*.

The **sizeimage** operator provides values for the *pixelsWide*, *pixelsHigh*, and *bps* parameters and for these device-dependent values:

- The number of color samples per pixel—1 (gray scale), 3 (RGB), or 4 (CMYK). If there's also an alpha component, you'll need to add 1 to this number to obtain *spp*.
- A Boolean value that reflects whether samples are meshed within a single data channel. If they're not meshed, the operator returns *true* in a *multiproc* parameter, indicating that in the PostScript language multiple procedures would be required to read the various samples.

NXInitHashState() → See NXHashInsert()

NXInsetRect() → See NXSetRect()

NXIntegralRect() → See NXSetRect()

NXIntersectionRect() → See NXUnionRect()

NXIntersectsRect() → See NXMouseInRect()

NXIsAlNum() → See NXIsAlpha()

NXIsAlpha(), NXIsAlNum(), NXIsCntrl(), NXIsDigit(), NXIsGraph(), NXIsLower(), NXIsPrint(), NXIsPunct(), NXIsSpace(), NXIsUpper(), NXIsXDigit(), NXIsAscii()

SUMMARY Classify NeXTstep–encoded values

LIBRARY libsys_s.a

SYNOPSIS

#import <NXCType.h>

int **NXIsAlpha**(unsigned *c*)
int **NXIsAlNum**(unsigned *c*)
int **NXIsUpper**(unsigned *c*)
int **NXIsLower**(unsigned *c*)
int **NXIsDigit**(unsigned *c*)
int **NXIsXDigit**(unsigned *c*)
int **NXIsSpace**(unsigned *c*)
int **NXIsPunct**(unsigned *c*)
int **NXIsPrint**(unsigned *c*)
int **NXIsGraph**(unsigned *c*)
int **NXIsCntrl**(unsigned *c*)
int **NXIsAscii**(unsigned *c*)

DESCRIPTION

These functions classify NeXTstep–encoded integer values. They return a nonzero value if the tested value belongs to the indicated class of characters or 0 if it does not.

These functions are similar to the standard C library routines for testing ASCII–encoded integer values (see the UNIX manual page for ctype), except that they act on the extended character set defined by NeXTstep encoding. For example, both **isalpha()** and **NXIsAlpha**() classify the character "a" as a letter; however, only **NXIsAlpha**() classifies "â" as a letter. The functions make these tests:

Function	**Tests that *c* is:**
NXIsAlpha(*c*)	a letter
NXIsUpper(*c*)	an uppercase letter
NXIsLower(*c*)	a lowercase letter
NXIsDigit(*c*)	a digit
NXIsXDigit(*c*)	a hexadecimal digit
NXIsAlNum(*c*)	an alphanumeric character
NXIsSpace(*c*)	a space, tab, carriage return, newline, vertical tab, or formfeed
NXIsPunct(*c*)	a punctuation character (neither control nor alphanumeric)
NXIsPrint(*c*)	a printing character
NXIsGraph(*c*)	a printing character; like **NXIsPrint()** except false for space
NXIsCntrl(*c*)	a control character (0x00 through 0x1F, 0x7F, 0x80, 0xFE, 0xFF)
NXIsAscii(*c*)	an ASCII character (code less than 0x7F)

RETURN

Each of these functions returns a nonzero value if the tested value belongs to the indicated class of characters or 0 if it does not.

SEE ALSO

NXToAscii()

NXIsAscii() → See NXIsAlpha()

NXIsCntrl() → See NXIsAlpha()

NXIsDigit() → See NXIsAlpha()

NXIsGraph() → See NXIsAlpha()

NXIsLower() → See NXIsAlpha()

NXIsPrint() → See NXIsAlpha()

NXIsPunct() → See NXIsAlpha()

NXIsServicesMenuItemEnabled() → See NXSetServicesMenuItemEnabled()

NXIsSpace() → See NXIsAlpha()

NXIsUpper() → See NXIsAlpha()

NXIsXDigit() → See NXIsAlpha()

NXJournalMouse()

SUMMARY	Allow journaling during direct mouse tracking
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <appkit/NXJournaler.h>

void **NXJournalMouse**(void)

DESCRIPTION

This function lets an application that accesses the status of the mouse directly (by calling functions such as **PSstilldown()** or **PScurrentmouse()**) participate in event journaling. If your application tests the status of the mouse by analyzing event records received through the Application Kit's normal distribution mechanism, you won't need to call this function.

For an application to be journaled, it must ask for events. If a routine in your application bypasses the Kit's event distribution system to test the mouse's position or button status, it must call **NXJournalMouse()** to ensure that its activities can be journaled. For example, a routine that takes some action as long as the mouse button is depressed should call **NXJournalMouse()** before testing the mouse:

```
do {
    NXJournalMouse();
    PSstilldown(mouseDownEvent.data.mouse.eventNum, &stillDown);
    /* Do some action */
} while (stillDown);
```

NXJournalMouse() asks for a journal-event, mouse-up, or mouse-dragged event, sends a copy to the journaler (if one is recording), and then discards the event.

Note: In the example above, releasing the mouse button causes the loop to exit. If the loop didn't call **NXJournalMouse()**, the mouse-up event would remain in the event queue after the loop exited. With the addition of **NXJournalMouse()**, this event is discarded. For most applications, this difference is of no consequence.

SEE ALSO

NXGetOrPeekEvent()

NXLogError()

SUMMARY	Write a formatted error string
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <appkit/nextstd.h>

void **NXLogError**(const char **format*, ...)

DESCRIPTION

NXLogError() is much like **printf**(). It writes a formatted string to the Console or **stderr**, depending on whether the application was launched from the Workspace Manager or some shell. **NXLogError**() calls **syslog**(), which marks the message with the time of occurrence and the application's process identification number. See the UNIX manual page for **syslog**() for more information.

SEE ALSO

NX_RAISE(), **NXDefaultExceptionRaiser**(), **NXRegisterErrorReporter**()

NXMagentaComponent() → See NXRedComponent()

NXMallocCheck() → See NXZoneMalloc()

NXMapFile() → See NXOpenMemory ()

NXMergeZone() → See NXZoneMalloc()

NXMouseInRect(), NXPointInRect(), NXIntersectsRect(), NXContainsRect(), NXEqualRect(), NXEmptyRect()

SUMMARY Test graphic relationships

LIBRARY libNeXT_s.a

SYNOPSIS

#import <appkit/graphics.h>

BOOL **NXMouseInRect**(const NXPoint **aPoint*, const NXRect **aRect*, BOOL *flipped*)
BOOL **NXPointInRect**(const NXPoint **aPoint*, const NXRect **aRect*)
BOOL **NXIntersectsRect**(const NXRect **aRect*, const NXRect **bRect*)
BOOL **NXContainsRect**(NXRect **aRect*, const NXRect **bRect*)
BOOL **NXEqualRect**(const NXRect **aRect*, const NXRect **bRect*)
BOOL **NXEmptyRect**(const NXRect **aRect*)

DESCRIPTION

These functions test the rectangles referred to by their arguments; they return YES if the test succeeds and NO if it fails. The functions that take two arguments assume that both arguments are expressed in the same coordinate system.

NXMouseInRect() is used to determine whether the hot spot of the cursor is inside a given rectangle. It returns YES if the point referred to by its first argument is located within the rectangle referred to by its second argument. If not, it returns NO. It assumes an unscaled and unrotated coordinate system.

The hot spot is the point within the cursor image that's used to report the cursor's location. It's situated at the upper left corner of a critical pixel in the cursor image, the one cursor pixel that's constrained to always be on screen. **NXMouseInRect**() is designed to return YES when this pixel is inside the rectangle, and NO when it's not. Thus if the point referred to by *aPoint* lies along the upper or left edge of the rectangle, this function should return YES. But if the point lies along the lower or right edge of the rectangle, it should return NO. To make this determination, the function needs to know the polarity of the y-axis. The third argument, *flipped*, should be NO if the positive y-axis extends upward, and YES if the coordinate system has been flipped so that the positive y-axis extends downward. (For convenience, View's **mouse:inRect:** method automatically determines whether the coordinate system is flipped. See the View class specification in Chapter 2 for more information about this method.)

NXPointInRect() performs the same test as **NXMouseInRect**() but assumes a flipped coordinate system. If the coordinate system is unflipped, it gives the wrong result if the point is coincident with the maximum or minimum y-coordinate of the rectangle. You should use **NXMouseInRect**() when testing the cursor's location.

NXContainsRect() returns YES if *aRect* completely encloses *bRect*. Otherwise, it returns NO.

NXIntersectsRect() returns YES if the two rectangles overlap, and NO otherwise. Adjacent rectangles that share only a side are not considered to overlap.

It's possible for **NXIntersectsRect()** to return NO even though the two rectangles include some of the same pixels. This can happen when the rectangles don't have any area in common, yet their outlines pass through some of the same pixels—for example, when they share a side not at a pixel boundary. In the NeXT imaging model, any pixel an outline passes through is treated as if it were inside the outline.

NXEqualRect() returns YES if the two rectangles are identical, and NO otherwise.

NXEmptyRect() returns YES if the rectangle encloses no area at all—that is, if it has no height or no width (or if its width or height is negative). If the height and width are both positive, it returns NO.

RETURN

These functions all return YES to indicate that the test succeeded and NO to indicate that it did not.

SEE ALSO

NXUnionRect(), **NXSetRect()**

NXNameObject() → See NXGetNamedObject()

NXNameZone() → See NXZoneMalloc()

NXNextHashState() → See NXHashInsert()

NXNoEffectFree() → See NXCreateHashTable()

NXNumberOfColorComponents() → See NXColorSpaceFromDepth()

NXOffsetRect() → See NXSetRect()

NXOpenFile(), NXOpenPort()

SUMMARY	Open a file stream or a Mach port stream
LIBRARY	libsys_s.a

SYNOPSIS

#import <streams/streams.h>

NXStream ***NXOpenFile**(int *fd*, int *mode*)
NXStream ***NXOpenPort**(port_t *port*, int *mode*)

DESCRIPTION

These functions connect a stream to a file or a Mach port. (The NXStream structure is defined in the header file **streams/streams.h**.)

NXOpenFile() opens a stream on the file specified by the file descriptor argument, *fd*, which can refer to a pipe or a socket. (If the file is stored on disk, use **NXMapFile()**; this function is described below under **NXOpenMemory()**.) The *mode* argument should be one of the three constants NX_READONLY, NX_WRITEONLY, or NX_READWRITE to specify how the stream will be used. The mode should be the same as the one used when obtaining the file descriptor. (The system call **open()**, which returns a file descriptor, takes 0_RDONLY, 0_WRONLY, or 0_RDWR to indicate whether the file will be used for reading, writing, or both. For more information on this function, see its UNIX manual page.)

You can use **NXOpenFile()** to connect to **stdin**, **stdout**, and **stderr** by obtaining their file descriptors using the standard C library function **fileno()**. (For more information on this function, see its UNIX manual page.)

NXOpenPort() opens a stream associated with the Mach port specified by *port*. The *mode* must be either NX_READONLY or NX_WRITEONLY. The port must already be allocated using the Mach function **port_allocate()**. See the "Mach Functions" section later in this chapter for more information about using this function.

Once the file or Mach port stream is open, you can read from or write to it. See the descriptions of **NXRead()** and **NXPutc()** for more information about the functions available for reading or writing to a stream.

When you're finished with the stream, close it with **NXClose()**. If you've written to the stream, the data will be automatically saved in the file. After calling **NXClose()** on a file stream, you still need to close the file descriptor. To do this, use the system call **close()**, giving it the file descriptor as an argument. (For more information about **close()**, see its UNIX manual page.)

RETURN

Both functions return a pointer to the stream they open or NULL if an error occurred while trying to open the stream.

SEE ALSO

NXOpenMemory(), **NXRead()**, **NXPutc()**, **NXClose()**

NXOpenMemory(), NXMapFile(), NXSaveToFile(), NXGetMemoryBuffer(), NXCloseMemory()

SUMMARY	Manipulate a memory stream
LIBRARY	libsys_s.a

SYNOPSIS

#import <streams/streams.h>

NXStream ***NXOpenMemory**(const char **address*, int *size*, int *mode*)
NXStream ***NXMapFile**(const char **pathName*, int *mode*)
int **NXSaveToFile**(NXStream **stream*, const char **name*)
void **NXGetMemoryBuffer**(NXStream **stream*, char ***streambuf*, int **len*,
    int **maxlen*)
void **NXCloseMemory**(NXStream **stream*, int *option*)

DESCRIPTION

These functions open, save, and close streams on memory. (The NXStream structure is defined in the header file **streams/streams.h**.)

NXOpenMemory() returns a pointer to the memory stream it opens. Its argument *mode* specifies whether the stream will be used for reading or writing. If NX_WRITEONLY is specified, the first two arguments should be NULL and 0 to allow the amount of memory available to be automatically adjusted as more data is written. Any other value for *address* should be the starting address of memory allocated with **vm_allocate()**. If NX_READONLY is specified, a memory stream will be set up for reading the data beginning at the location specified by the first argument; the second argument indicates how much data will be read. To use the stream for both writing and reading, you can either use NULL and 0 or specify the location and amount of data to be read; again, *address* should be the starting address of memory allocated with **vm_allocate()**.

NXMapFile() maps a file into memory and then opens a memory stream. A related function, **NXOpenFile()**, connects a stream to a file specified with a file descriptor. (This function is described earlier in this chapter.) Memory mapping allows efficient random and multiple access to the data in the file, so **NXMapFile()** should be used whenever the file is stored on disk. When you call **NXMapFile()**, give it the pathname for the file and indicate whether you will be writing, reading, or both, by using one of the *mode* constants described above. If you use the stream only for reading, just close the memory stream when you're finished. If you write to the memory-mapped stream, you need to call **NXSaveToFile()**, as described below, to save the data.

Once the memory stream is open, you can read from or write to it. See the descriptions of **NXRead()** and **NXPutc()** for more information about reading or writing to a stream.

Before you close a memory stream, you can save data written to the stream in a file. To do this, call **NXSaveToFile()**, giving it the stream and a pathname as arguments. **NXSaveToFile()** writes the contents of the memory stream into the file, creating it if necessary. After saving the data, close the stream using **NXCloseMemory()**.

NXGetMemoryBuffer() returns the memory buffer (*streambuf*) and its current and maximum lengths (*len* and *maxlen*).

When you're finished with a memory stream, close it by calling **NXCloseMemory()**. Typically, NX_FREEBUFFER will be used as the second argument to free all memory used by the stream, but there are two other constants available. If you've used the stream for writing, more memory may have been made available than was actually used; the constant NX_TRUNCATEBUFFER indicates that any unused pages of memory should be freed. (Calling **NXClose()** with a memory stream is equivalent to calling **NXCloseMemory()** and specifying NX_TRUNCATEBUFFER.) NX_SAVEBUFFER doesn't free the memory that had been made available.

RETURN

NXOpenMemory() and **NXMapFile()** return a pointer to the stream they open or NULL if the stream couldn't be opened.

NXSaveToFile() returns –1 if an error occurred while opening or writing to the file and 0 otherwise.

EXCEPTIONS

The functions in this group that take a stream as an argument raise an NX_illegalStream exception if the stream is invalid. This exception is also raised if these functions are used on a stream that isn't a memory stream.

SEE ALSO

NXRead(), NXPutc(), NXOpenFile()

NXOpenPort() → See NXOpenFile()

NXOpenTypedStream(), NXCloseTypedStream(), NXOpenTypedStreamForFile()

SUMMARY Open or close a typed stream

LIBRARY libsys_s.a

SYNOPSIS

#import <objc/typedstream.h>

NXTypedStream ***NXOpenTypedStream**(NXStream **stream*, int *mode*)
void **NXCloseTypedStream**(NXTypedStream **typedStream*)
NXTypedStream ***NXOpenTypedStreamForFile**(const char **fileName*, int *mode*)

DESCRIPTION

These functions open, save the contents of, and close a typed stream. A typed stream should be used for archiving—that is, for saving Objective-C objects for later use, typically in a file. (The NXTypedStream type is declared in the header file **objc/typedstream.h**. The structure itself is private since you never need to access its members.)

The first argument for **NXOpenTypedStream**() is an already opened NXStream structure. See the descriptions of **NXOpenMemory**(), **NXOpenFile**(), and **NXOpenPort**() earlier in this chapter for more information about opening a stream. The second argument to **NXOpenTypedStream**() must be NX_READONLY or NX_WRITEONLY to specify how the typed stream will be used.

Once the typed stream is open, you can write to or read from it. See the descriptions of **NXReadType**(), **NXReadObject**(), and **NXReadPoint**() later in this chapter for more information about reading and writing. When you're finished with the typed stream, you must first close the typed stream using **NXCloseTypedStream()** and then close the NXStream structure. See the descriptions of **NXClose**() and **NXCloseMemory**() for more information about closing a stream.

To open a typed stream on a file, use **NXOpenTypedStreamForFile**(). This function opens a memory stream and an associated typed stream. If *mode* is NX_READONLY, the typed stream is initialized with the contents of the file specified by *fileName*. A subsequent call to **NXCloseTypedStream()** will close the NXTypedStream and NXStream structures and free the buffer that had been used. If *mode* is NX_WRITEONLY, a typed stream on memory is opened, ready for writing. When you finish writing, calling **NXCloseTypedStream**() will flush the typed stream, save its contents in the file specified by *fileName*, close both the NXTypedStream and the NXStream structures, and free the buffer used.

RETURN

NXOpenTypedStream() and **NXOpenTypedStreamForFile**() return a pointer to the typed stream they open or NULL if the stream couldn't be opened.

EXCEPTIONS

NXOpenTypedStream() and **NXOpenTypedStreamForFile**() raise a TYPEDSTREAM_CALLER_ERROR exception with the message "NXOpenTypedStream: invalid mode" if the mode is anything other than NX_READONLY or NX_WRITEONLY.

NXOpenTypedStream() raises a TYPEDSTREAM_CALLER_ERROR exception with the message "NXOpenTypedStream: null stream" if an invalid NXStream structure is passed in.

SEE ALSO

NXOpenMemory(), **NXOpenFile**(), **NXClose**(), **NXCloseMemory**(), **NXReadType**(), **NXReadObject**(), **NXReadPoint**()

NXOpenTypedStreamForFile() → See NXOpenTypedStream()

NXOrderStrings(), NXDefaultStringOrderTable()

SUMMARY Provide table-driven string ordering service

LIBRARY libNeXT_s.a

SYNOPSIS

#import <appkit/Text.h>

int **NXOrderStrings**(const unsigned char **s1*, const unsigned char **s2*,
    BOOL *caseSensitive*, int *length*, NXStringOrderTable **table*)
NXStringOrderTable ***NXDefaultStringOrderTable**(void)

DESCRIPTION

NXOrderStrings() returns a value indicating the ordering of the strings *s1* and *s2*, as determined by the NXStringOrderTable structure *table*. If *caseSensitive* is NO, capital and lowercase versions of a letter are considered to have identical rank. The comparison considers at most the first *length* characters of each string. For convenience, you can pass –1 for *length* if both strings are null-terminated. If *table* is NULL, the default ordering table (as described below) is used. **NXOrderStrings**() returns 1, 0, or –1 depending on whether *s1* is greater than, equal to, or less than *s2* according to *table*.

When comparing strings that are visible to the user, you should generally use **NXOrderStrings**(*s1*, *s2*, YES, –1, NULL) as a replacement for **strcmp**(*s1*, *s2*) and **NXOrderStrings**(*s1*, *s2*, YES, *n*, NULL) as a replacement for **strncmp**(*s1*, *s2*, *n*).

NXOrderStrings() consults an NXStringOrderTable structure when comparing strings. This structure is declared in **appkit/Text.h**:

```
typedef struct {
    unsigned char primary[256];
    unsigned char secondary[256];
    unsigned char primaryCI[256];
    unsigned char secondaryCI[256];
} NXStringOrderTable;
```

The first two arrays contain ordering information for case sensitive searches; the last two are for case insensitive searches. **NXOrderStrings()** determines a character's rank by using the character to index into the appropriate primary array. The value found at that position determines the character's rank. For example, in the default ordering table the value at the 'a' position is less than that at the 'b' position, but the values at the 'o' and 'ö' positions are identical. The secondary arrays provide additional ordering information for ligature characters (such as 'æ' and 'fl'), in effect breaking the ligature apart for the purposes of ordering. Thus, the two characters 'ae' and the single character 'æ' are given equal rank.

NeXTstep provides a default order table, which can by accessed by calling **NXDefaultStringOrderTable()**. If you want to create your own order table, it's best to start with the default table and algorithmically modify it (perhaps in conjunction with the NXCType routines—see **/usr/include/NXCTypes.h**). In this way, you'll benefit from using character tables that have already been localized. The entry at the 0 position in each array must be 0.

RETURN

NXOrderStrings() returns 1, 0, or –1 depending on whether *s1* is greater than, equal to, or less than *s2* according to *table*. **NXDefaultStringOrderTable()** returns a pointer to the default string order table.

NXPing()

SUMMARY Synchronize the application with the Window Server

LIBRARY libNeXT_s.a

SYNOPSIS

#import <appkit/graphics.h>

void **NXPing**(void)

DESCRIPTION

NXPing() helps applications synchronize their actions with the actions of the Window Server; it enables an application to respond smoothly to user events.

An application can generate PostScript code faster than the Window Server can interpret it. An application can therefore "get ahead" of the Server—it can get events and respond to them before its responses to previous events are displayed to the user. To the user, it appears that the application is slow, or that there's discontinuity between an event and the response.

NXPing() causes the application to pause until the Window Server catches up. It flushes the connection buffer so that all current PostScript code is sent to the Server and returns only when all the code has been interpreted. It's a cover for the **DPSWaitContext()** function when passed the context returned by **DPSGetCurrentContext()**:

```
DPSWaitContext(DPSGetCurrentContext())
```

For more information on these two Display PostScript functions, see the *Client Library Reference Manual.*

Waiting for the Window Server to catch up with the application is sometimes a good idea, for two reasons:

- It lets the Server have full access to the CPU. The application stops competing with it for system resources.
- It gives the application a chance to generate less, and more relevant, PostScript code. An application won't fall even further behind the user while it waits for the Window Server if it combines its responses to events or allows events to be coalesced in the event queue.

NXPing() is most typically used in a modal loop. In a tracking loop, it should be called just before getting each new event (after all the PostScript code has been generated in response to the last event). The following schematic for a **mouseDown:** method illustrates its use. (Comments that would be replaced by code in any real method are shown in italic type.)

```
- mouseDown:(NXEvent *)thisEvent
{
    BOOL   shouldLoop = YES;
    int    oldMask = [window addToEventMask:NX_LMOUSEDRAGGEDMASK];

    while ( shouldLoop ) {
        /*
         * Draw in response to the event
         */
        NXPing();
        theEvent = [NXApp getNextEvent:(NX_LMOUSEUPMASK
                                        | NX_LMOUSEDRAGGEDMASK)];
        if ( theEvent->type == NX_LMOUSEUP )
            shouldLoop = NO;
    }
    /*
     * Replace dynamic drawing with a static display
     */
    [window setEventMask:oldMask];
    return self;
}
```

During the wait imposed by **NXPing()**, mouse-dragged (and mouse-moved) events will be coalesced in the event queue. When the application next gets an event, it will be a more up-to-date one than if **NXPing()** had not been used. Coalescing also serves to reduce the total amount of PostScript code generated.

NXPing() also lets an application more efficiently group its responses to a number of similar events. In the following example, the method that responds to key-down events uses the **peekNextEvent:into:** method to take all available key-down events from the event queue and display them at once. The use of **NXPing()** means that the example will be invoked less often than it otherwise would. However, it will consolidate events into fewer instructions for the Window Server.

```
- keyDown:(NXEvent *)theEvent
{
    /*
     * Check theEvent->data.key.charSet and
     * theEvent->data.key.charCode and set up the array of
     * characters to displayed
     */
    while ( 1 ) {
        /* Peek at the next event */
        NXEvent next;
        theEvent = [NXApp peekNextEvent:NX_ALLEVENTS into:&next];
        /* Break the loop if there is no next event */
        if ( !theEvent )
            break;
        /* Skip over key-up events */
        else if ( theEvent->type == NX_KEYUP ) {
            [NXApp getNextEvent:NX_KEYUPMASK];
            continue;
        }
        /* Respond only to key-down events */
        else if ( theEvent->type == NX_KEYDOWN ) {
            /*
             * Add the new character to the array to be displayed
             */
            [NXApp getNextEvent:NX_KEYDOWNMASK];
        }
        /* Break the loop on all other events types */
        else
            break;
    }
    /*
     * Display the array of characters
     */
    NXPing();
    return self;
}
```

The wait imposed by **NXPing()** may mean that there are more key-down events in the event queue each time this method is invoked. Since it's much more efficient for the application to send fewer instructions to the Window Server to display longer strings, this delay helps rather than hurts.

In the examples shown above, **NXPing()** is called just before the application is ready to get another event. This is the most appropriate place for it, since it means that the response to the last event will be complete—including the Window Server's part—before the response to the next event begins. It might be noted that both **NXPing()** and the functions and methods that get events flush the output buffer to the Window Server. However, the buffer isn't flushed if it's empty, so calling **NXPing()** before getting an event doesn't cause an extra operation to be performed.

Using **NXPing()** has two negative consequences:

- It reduces the Window Server's throughput—the amount of PostScript code that it can interpret in a given time period. This is mainly due to the increased communication between the Server and the application.

- It reduces the granularity of the application's response to events. When events are coalesced in the event queue, cursor movements are tracked at greater intervals.

Therefore, you should not use **NXPing()** in a simple event loop unless the time needed to execute the PostScript code each event generates is longer than the time needed to complete the loop.

Although **NXPing()** is most often used in modal loops, it's also appropriate to use it in situations where information from the Window Server is needed before the application can proceed. For example, you may want to call **NXPing()** before entering a section of code that depends on previous PostScript instructions being executed without error. Since your application won't get notified of any errors until the PostScript code is actually executed, **NXPing()** allows it to wait for the notification before proceeding.

SEE ALSO

DPSFlush()

NXPointInRect() → See NXMouseInRect()

NXPortFromName(), NXPortNameLookup()

SUMMARY	Get send rights to an application port
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <appkit/ Listener.h>

port_t **NXPortFromName**(const char **name*, const char **host*)
port_t **NXPortNameLookup**(const char **name*, const char **host*)

DESCRIPTION

NXPortFromName() and **NXPortNameLookup()** both return send rights to the port that's registered with the Network Name Server under *name* for the *host* machine. If *host* is a NULL pointer or an empty string, the local host is assumed. This is the most common usage.

An application generally registers with the Network Name Server under the name it uses for its executable file. For example, Digital Webster™ registers under "Webster" and Mail under "Mail".

If no port is registered for the *name* application, **NXPortNameLookup()** returns PORT_NULL. However, **NXPortFromName()** tries to have *host*'s Workspace Manager launch the application. If the application can be launched and if it registers with the Network Name Server, send rights to its port are returned. This strategy is almost always successful for the local host. It's more problematic for a remote host, since the Workspace Manager is normally protected from messages coming from other machines.

If, in the end, no port can be found for the *name* application, **NXPortFromName()**, like **NXPortNameLookup()**, returns PORT_NULL.

Applications should use these two functions, rather than the Mach **netname_look_up()** function, to get send rights to a public port. Although both functions currently use **netname_look_up()** to find the port, this may not always be true. In future releases, Listener objects may "check in" with another service—such as the Bootstrap Server—rather than the Network Name Server. In this case, the two functions described here will continue to find and return the port associated with *name*, but **netname_look_up()** will not.

RETURN

Both functions return send rights to the public port of the *name* application on the *host* machine, or PORT_NULL if the port can't be found.

NXPortNameLookup() → See NXPortFromName()

NXPrintf() → See NXPutc()

NXPtrHash() → See NXCreateHashTable()

NXPtrIsEqual() → See NXCreateHashTable()

NXPutc(), NXGetc(), NXUngetc(), NXScanf(), NXPrintf(), NXVScanf(), NXVPrintf()

SUMMARY	Read or write formatted data to or from a stream
LIBRARY	libsys_s.a

SYNOPSIS

#import <streams/streams.h>

int **NXPutc**(NXStream **stream*, char *c*)
int **NXGetc**(NXStream **stream*)
void **NXUngetc**(NXStream **stream*)
int **NXScanf**(NXStream **stream*, const char **format*, ...)
void **NXPrintf**(NXStream **stream*, const char **format*, ...)
int **NXVScanf**(NXStream **stream*, const char **format*, va_list *argList*)
void **NXVPrintf**(NXStream **stream*, const char **format*, va_list *argList*)

DESCRIPTION

These functions and macros read and write data to and from a stream that has already been opened. (See the descriptions of **NXOpenMemory**() and **NXOpenFile**() for more information about opening a stream.) After writing to a stream, you may need to call **NXFlush**() to flush data from the buffer associated with the stream. (See the description of **NXFlush**() earlier in this chapter.)

The macros for writing and reading single characters at a time are similar to the corresponding standard C functions: **NXPutc**() and **NXGetc**() work like **putc**() and **getc**(). **NXPutc**() appends a character to the stream. Its second argument specifies the character to be written to the stream. **NXGetc**() retrieves the next character from the stream. To reread a character, call **NXUngetc**(). This function puts the last character read back onto the stream. **NXUngetc**() doesn't take a character as an argument as **ungetc**() does. **NXUngetc**() can only be called once between any two calls to **NXGetc**() (or any other reading function).

The other four functions convert strings of data as they're written to or read from a stream. **NXPrintf**() and **NXScanf**() take a character string that specifies the format of the data to be written or read as an argument. **NXPrintf**() interprets its variables according to the format string and writes them to the stream. Similarly, **NXScanf**() reads characters from the stream, interprets them as specified in the format string, and stores them in the variables indicated by the last set of arguments. The conversion characters in the format string for both functions are the same as those used for the standard C library functions, **printf**() and **scanf**(). For detailed information on these characters and how conversions are performed, see the UNIX manual pages for **printf**() and **scanf**().

Two related functions, **NXVPrintf** () and **NXVScanf()**, are exactly the same as **NXPrintf()** and **NXScanf()**, except that instead of being called with a variable number of arguments, they are called with a **va_list** argument list, which is defined in the header file **stdarg.h**. This header file also defines a set of macros for advancing through a **va_list**.

RETURN

NXPutc() and **NXGetc()** return the character written or read. **NXScanf()** and **NXVScanf()** return EOF if all data was successfully read; otherwise, they return the number of successfully read data items.

SEE ALSO

NXOpenMemory(), NXOpenFile(), NXFlush(), NXRead()

NXRead(), NXWrite()

SUMMARY Read from or write to a stream

LIBRARY libsys_s.a

SYNOPSIS

#import <streams/streams.h>

int **NXRead**(NXStream **stream*, void **buf*, int *count*)
int **NXWrite**(NXStream **stream*, const void **buf*, int *count*)

DESCRIPTION

These functions read and write multiple bytes of data to a stream that has already been opened. (See the descriptions of **NXOpenMemory()** and **NXOpenFile()** for more information about opening a stream.) After writing to a stream, you may need to call **NXFlush()** to flush data from the buffer associated with the stream. (See the description of **NXFlush()** earlier in this chapter.)

These functions write multiple bytes of data to and read them from a stream. To read data from a stream, call **NXRead()**:

```
NXRect        myRect;
NXRead(stream, &myRect, sizeof(NXRect));
```

NXRead() reads the number of bytes specified by its third argument from the given stream and places the data in the location specified by the second argument.

In the following example, an NXRect structure is written to a stream.

```
NXRect  myRect;

NXSetRect(&myRect, 0.0, 0.0, 100.0, 200.0);
NXWrite(stream, &myRect, sizeof(NXRect));
```

The second and third arguments for **NXWrite()** give the location and amount of data (measured in bytes) to be written to the stream.

RETURN

These functions return the number of bytes written or read. If an error occurs while writing or reading, not all the data will be written or read.

SEE ALSO

NXFlush()

NXReadArray(), NXWriteArray()

SUMMARY Read or write arrays from or to a typed stream

LIBRARY libsys_s.a

SYNOPSIS

#import <objc/typedstream.h>

void **NXReadArray**(NXTypedStream **typedStream*, const char **dataType*, int *count*, const void **data*)
void **NXWriteArray**(NXTypedStream **typedStream*, const char **dataType*, int *count*, void **data*)

DESCRIPTION

These functions read and write arrays from and to a typed stream. They can be used within **read:** or **write:** methods for archiving purposes. See the description of **NXReadObject()** in this chapter for more about these methods. Functions are also available for reading and writing other data types; they're listed below under "SEE ALSO."

Before using a typed stream for reading and writing, it must be opened; see the description of **NXOpenTypedStream()** for details on opening a typed stream. (The NXTypedStream type is declared in the header file **objc/typedstream.h**. The structure itself is private since you never need to access its members.)

NXReadArray() and **NXWriteArray()** read and write an array of *count* elements of type *dataType* from or to *typedStream*. **NXReadArray()** reads the array from the typed stream into the location specified by *data*, which must have been previously allocated. **NXWriteArray()** writes the array specified by *data* to the typed stream. Both functions use the characters listed under the description of **NXReadType()** for *dataType*.

The following is an example of an integer array being written. To read the same array, **NXReadArray()** would be called with the same first three arguments as **NXWriteArray()**; the fourth argument would be a pointer to memory for the array.

```
int  aa[4];

aa[0] = 0; aa[1] = 11; aa[2] = 22; aa[3] = 33;
NXWriteArray(typedStream, "i", 4, aa);
```

EXCEPTIONS

Both functions check whether the typed stream has been opened for reading or for writing and raise a TYPEDSTREAM_FILE_INCONSISTENCY exception if it isn't correct. For example, if **NXReadArray()** is called and the stream was opened for writing, the exception is raised.

NXReadArray() raises a TYPEDSTREAM_FILE_INCONSISTENCY exception if the data to be read is not of the expected type.

SEE ALSO

NXOpenTypedStream(), **NXReadType**(), **NXReadObject**(), and **NXReadPoint**()

NXReadBitmap() → See NXImageBitmap()

NXReadColor(), NXWriteColor()

SUMMARY	Read and write a color from a typed stream
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <appkit/color.h>

NXColor **NXReadColor**(NXTypedStream **stream*)
void **NXWriteColor**(NXTypedStream **stream*, NXColor *color*)

DESCRIPTION

NXReadColor() reads a color from the typed stream, *stream*, and returns it. **NXWriteColor**() writes a color value, *color*, to a typed stream. The stream can be connected to a file, to memory, or to some other repository for data.

NXColor values should be read and written only using these functions. When a color is written by **NXWriteColor**() and then read back by **NXReadColor**(), the color is guaranteed to be the same. This cannot be guaranteed if NXColor structures are read and written directly—for example, through standard C functions like **fread**() and **fwrite**(). The internal format of an NXColor data structure is not specified and therefore may change in future releases.

RETURN

NXReadColor() returns the color value it reads.

EXCEPTION

NXReadColor() raises an NX_newerTypedStream exception if the data it's expected to read is not of type NXColor.

SEE ALSO

NXSetColor(), **NXConvertRGBAToColor**(), **NXConvertColorToRGBA**(), **NXEqualColor**(), **NXRedComponent**(), **NXChangeRedComponent**()

NXReadDefault() → See NXRegisterDefaults()

NXReadObject(), NXWriteObject(), NXWriteObjectReference(), NXWriteRootObject()

SUMMARY Read or write Objective-C objects from or to a typed stream

LIBRARY libsys_s.a

SYNOPSIS

#import <objc/typedstream.h>

id **NXReadObject**(NXTypedStream **typedStream*)
void **NXWriteObject**(NXTypedStream **typedStream*, id *object*)
void **NXWriteObjectReference**(NXTypedStream **typedStream*, id *object*)
void **NXWriteRootObject**(NXTypedStream **typedStream*, id *rootObject*)

DESCRIPTION

These functions initiate the archiving and unarchiving processes for Objective-C objects. They read and write the object passed in from or to *typedStream*. When an object is archived with these functions, its class is automatically written as well. In addition, the data type of each of its instance variables is archived along with the value of the variable. These functions also ensure that objects are written only once.

Before you use a typed stream for reading and writing, it must be opened; see the description of **NXOpenTypedStream()** for details on opening a typed stream. (The NXTypedStream type is declared in the header file **objc/typedstream.h**. The structure itself is private since you never need to access its members.)

NXReadObject() begins the unarchival process by allocating memory for a new object of the correct class. It then sends the object a **read:** message to initialize its instance variables from the typed stream. **read:** messages should only be generated through **NXReadObject**(); they shouldn't be sent directly to objects. Application Kit objects already have **read:** methods, but you need to implement **read:** methods for any classes you create that add instance variables:

```
- read:(NXTypedStream *)typedStream
{
    [super read:typedStream];
    . . ./* code for reading instance variables declared in
            this class */
}
```

The message to **super** ensures that inherited instance variables will be unarchived. The body of the **read:** method unarchives the object's instance variables, using the appropriate function for that data type. The functions available for unarchiving include

NXReadTypes(), **NXReadPoint()**, and **NXReadArray()**, as well as **NXReadObject()**. See the descriptions of these functions in this chapter for information about how to use them. A **read:** method can also check the version of the class being unarchived. See the description of **NXTypedStreamClassVersion()** for more information about how to do this.

After **NXReadObject()** unarchives an object, it sends the object **awake** and **finishUnarchiving** messages. You can implement an **awake** method to initialize the object to a usable state. The **finishUnarchiving** method allows you to replace the just-unarchived object with another one. If you implement a **finishUnarchiving** method, it should free the unarchived object and return the replacement object.

NXWriteObject() writes *object* to *typedStream* by sending the object a **write:** message. As is the case with **read:** methods, **write:** methods shouldn't be sent directly to objects, and they need to be implemented for classes that add instance variables. They also need to begin with a message to **super**. The functions available for archiving instance variables parallel those for unarchiving; they include **NXWriteTypes()**, **NXWritePoint()**, and **NXWriteArray()**, all of which are described elsewhere in this chapter. If the object being archived has **id** instance variables (including those that are statically typed to a class), they're archived as described below.

In some cases, an object's **id** instance variables contain inherent properties of the object to which they belong, or they might be necessary for the object to be usable. For example, a View's subview list is an intrinsic part of that View, just as a ButtonCell is needed for a Button to work properly. For these kinds of instance variables, the object—the View or the Button in the examples mentioned—uses **NXWriteObject()** within its **write:** method. (Actually, Button objects inherit Control's **write:** method, which archives the **cell** instance variable.) The function **NXWriteTypes()** can also be used to archive **id** instance variables, by specifying the **id** data type format character.

In other cases, an object's **id** instance variables refer to other objects that act at the discretion of the object, such as its target or delegate, or that aren't inherently part of the object. A View's **superview** and **window** instance variables aren't considered intrinsic to the View since you might want to hook up the View to another superview or to a different Window. For these kinds of instance variables, the object calls **NXWriteObjectReference()** within its **write:** method. When archiving a data structure that includes objects that have called **NXWriteObjectReference()**, **NXWriteRootObject()** must be used instead of **NXWriteObject()**.

NXWriteObjectReference() specifies that a pointer to **nil** should be written for the object passed in, unless that object is an intrinsic part of some member of the data structure being archived. If the object is intrinsic, it will be archived and, after unarchiving, the pointer will point to the object. **NXWriteRootObject()** makes two passes through the data structure being written. The first time, it defines the limits of the data to be written by including instance variables intrinsic to the data structure and by making a note of which objects have been written with **NXWriteObjectReference()**. On the second pass, **NXWriteRootObject()** archives the data structure.

As an example, consider a View that has a Button as one subview and a TextField, which is the target of the Button, as another subview. If you archive the Button, its ButtonCell will be written. The archived ButtonCell's **target** instance variable will point to **nil**. If you archive the View, however, the Button and the TextField will be archived since they're subviews. The ButtonCell will be archived since it's needed by the Button. The ButtonCell's **target** instance variable will point to the TextField since it's an intrinsic part of the View.

RETURN

NXReadObject() returns the **id** of the object read.

EXCEPTIONS

All functions check whether the typed stream has been opened for reading or for writing and raise a TYPEDSTREAM_CALLER_ERROR exception with an appropriate message if it isn't correct. For example, if **NXReadObject**() is called and the stream was opened for writing, an exception is raised.

If an error occurs while creating an instance of the appropriate class, **NXReadObject**() raises a TYPEDSTREAM_CLASS_ERROR. This function also raises a TYPEDSTREAM_FILE_INCONSISTENCY exception if the data to be read is not of type **id**.

If **NXWriteObject**() is used to archive a data structure that includes objects with calls to **NXWriteObjectReference**(), a TYPEDSTREAM_WRITE_REFERENCE_ERROR exception is raised.

SEE ALSO

NXOpenTypedStream(), **NXReadArray**(), **NXReadType**(), **NXReadPoint**(), and **NXTypedStreamClassVersion**()

NXReadObjectFromBuffer(), NXReadObjectFromBufferWithZone(), NXWriteRootObjectToBuffer(), NXFreeObjectBuffer()

SUMMARY	Read and write an object to a typed-stream memory buffer
LIBRARY	libsys_s.a

SYNOPSIS

#import <objc/typedstream.h>

id **NXReadObjectFromBuffer**(const char **buffer*, int *length*)
id **NXReadObjectFromBufferWithZone**(const char **buffer*, int *length*,
    NXZone **zone*)
char ***NXWriteRootObjectToBuffer**(id *object*, int **length*)
void **NXFreeObjectBuffer**(char **buffer*, int *length*)

DESCRIPTION

These functions allow you to easily read and write an object to a typed stream on memory. They're particularly useful for archiving an object, writing it to the pasteboard, and then unarchiving it from the pasteboard.

NXWriteRootObjectToBuffer() opens a stream on memory (using **NXOpenMemory**()) and a corresponding typed stream. It then writes the object given as its argument by calling **NXWriteRootObject()** and closes the typed stream. (See the description of **NXWriteRootObject()** under **NXReadObject()** above for more information about how the object is written.) **NXWriteRootObjectToBuffer()** also closes the memory stream but retains the buffer, which is truncated to the size of the object. **NXWriteRootObjectToBuffer()** returns the size of the object (in the location specified by *length*) and a pointer to the buffer itself.

NXReadObjectFromBuffer() calls **NXReadObjectFromBufferWithZone()** with the default zone as its *zone* argument.

NXReadObjectFromBufferWithZone() opens a stream on memory and a corresponding typed stream with its zone set by the **NXSetTypedStreamZone**() function. The *buffer* and *length* arguments passed in should be taken from a previous call to **NXWriteRootObjectToBuffer**(). **NXReadObject**() is called to read the object from the buffer into the zone, after which the streams are closed. **NXReadObjecFromBufferWithZone()** saves the memory buffer and returns the object it reads in the zone specified. Unless you're going to reread the buffer, you should free it using the **NXFreeObjectBuffer**() function.

NXFreeObjectBuffer() frees the buffer specified by *buffer*, which should be *length* bytes long. These arguments should be taken from a previous call to **NXWriteRootObjectToBuffer**().

RETURN

NXReadObjectFromBuffer() returns the object it reads from the buffer.

NXWriteRootObjectToBuffer() returns a pointer to the buffer it creates.

EXCEPTIONS

NXReadObjectFromBuffer() and **NXReadObjectFromBufferWithZone**() raise a TYPEDSTREAM _FILE_INCONSISTENCY exception if the data to be read from the buffer is not of type **id**.

SEE ALSO

NXOpenMemory(), **NXReadObject**(), and **NXOpenTypedStream()**

NXReadObjectFromBufferWithZone() → NXReadObjectFromBuffer()

NXReadPoint(), NXWritePoint(), NXReadRect(), NXWriteRect(), NXReadSize(), NXWriteSize()

SUMMARY	Read or write NeXT-defined data types to a typed stream
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <appkit/graphics.h>

void **NXReadPoint**(NXTypedStream **typedStream*, NXPoint **aPoint*)
void **NXWritePoint**(NXTypedStream **typedStream*, const NXPoint **aPoint*)
void **NXReadRect**(NXTypedStream **typedStream*, NXRect **aRect*)
void **NXWriteRect**(NXTypedStream **typedStream*, const NXRect **aRect*)
void **NXReadSize**(NXTypedStream **typedStream*, NXSize **aSize*)
void **NXWriteSize**(NXTypedStream **typedStream*, const NXSize **aSize*)

DESCRIPTION

These functions read and write NXPoint, NXSize, or NXRect structures from and to a typed stream. They can be used within **read:** or **write:** methods for archiving purposes. See the description of **NXReadObject()** in this chapter for more about these methods. Functions are also available for reading and writing other data types; they're listed below under "SEE ALSO."

Before using a typed stream for reading and writing, it must be opened; see the description of **NXOpenTypedStream()** for details on opening a typed stream. (The NXTypedStream type is declared in the header file **objc/typedstream.h**. The structure itself is private since you never need to access its members.)

NXReadPoint(), **NXReadSize()**, and **NXReadRect()** take a typed stream as an argument and place the data read from the stream into the location specified by the second argument. They work through **NXReadType()**.

The three corresponding writing functions work through **NXWriteType()** to write the data specified by their second argument to the typed stream. Note that the second argument should be a pointer to the data. The following example shows the three kinds of structures being written to an already opened typed stream; to read the same data, the corresponding reading functions would be called with the same arguments.

```
NXPoint  zeroPoint = {0.0, 0.0};
NXSize   rectSize = {100.0, 200.0};
NXRect   aRect = {zeroPoint, rectSize};

NXWritePoint(stream, &zeroPoint);
NXWriteSize(stream, &rectSize);
NXWriteRect(stream, &aRect);
```

EXCEPTIONS

All six functions check whether the typed stream has been opened for reading or for writing and raise a TYPEDSTREAM_FILE_INCONSISTENCY exception if the type isn't correct. For example, if **NXReadPoint**() is called and the stream was opened for writing, the exception is raised.

The functions for reading raise a TYPEDSTREAM_FILE_INCONSISTENCY exception if the data to be read is not of the expected type.

SEE ALSO

NXOpenTypedStream(), **NXReadType**(), **NXReadArray**(), **NXReadObject**()

NXReadRect() → See NXReadPoint()

NXReadSize() → See NXReadPoint()

NXReadTIFF(), NXWriteTIFF(), NXGetTIFFInfo()

SUMMARY	Read and write TIFF files
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <appkit/tiff.h>

void ***NXReadTIFF**(int *imageNumber*, NXStream **stream*, NXTIFFInfo **info*, void **data*)
void **NXWriteTIFF**(NXStream **stream*, NXImageInfo **image*, void **data*)
int **NXGetTIFFInfo**(int *imageNumber*, NXStream **stream*, NXTIFFInfo **info*)

DESCRIPTION

These functions read and write image data that's been stored in a TIFF file. This file format is described in the *Tag Image File Format Specification, Revision 5.0*. (See "Suggested Reading" in the *Technical Summaries* manual for information about how to obtain the TIFF specification manual.)

All three functions take a pointer to an NXStream structure as an argument. This stream should be opened on a TIFF file. (The NXStream structure is defined in the header file **streams/streams.h**.)

NXReadTIFF() reads the image data for the image specified by *imageNumber* from the *stream*. The *info* argument points to an uninitialized NXTIFFInfo structure, which you should allocate on the stack. **NXReadTIFF**() calls **NXGetTIFFInfo**() to read the information that describes the image into the NXTIFFInfo structure. This structure is defined in the header file **appkit/tiff.h**. The image data will be stored in the memory pointed to by *data*. If *data* is NULL, memory for the image data will be made available using **malloc()**. If an error occurs while reading the data, the error field of the NXTIFFInfo structure will be nonzero, and **NXReadTIFF()** will return NULL.

NXWriteTIFF() writes an image to the *stream* so that it can be saved in a TIFF file. The NXImageInfo structure specified by *image* describes the image to be written, and *data* points to the image data to be written. The NXImageInfo structure is defined in **appkit/tiff.h**.

NXGetTIFFInfo() reads the information for the image specified by *imageNumber* from the stream. The information is stored in the uninitialized NXTIFFInfo structure pointed to by *info*, which you should allocate on the stack. This information provides enough detail so that you can read the image data when desired, for example to edit it programmatically. The total number of bytes for the image is returned unless there is an error. If an error occurs, the error field of the NXTIFFInfo structure will have a nonzero value and **NXGetTIFFInfo()** will return 0.

RETURN

NXReadTIFF() returns a pointer to the image data read unless an error occurs while reading, in which case it returns NULL.

NXGetTIFFInfo() returns the number of bytes needed to store the image or 0 if an error occurred while reading the image information.

NXReadType(), NXWriteType(), NXReadTypes(), NXWriteTypes()

SUMMARY	Read or write arbitrary data to a typed stream
LIBRARY	libsys_s.a

SYNOPSIS

#import <objc/typedstream.h>

void **NXReadType(**NXTypedStream **typedStream*, const char **type*, void **data***)**
void **NXWriteType(**NXTypedStream **typedStream*, const char **type*,
    const void **data***)**
void **NXReadTypes(**NXTypedStream **typedStream*, const char **types*, ...**)**
void **NXWriteTypes(**NXTypedStream **typedStream*, const char **types*, ...**)**

DESCRIPTION

These functions read and write strings of data from and to a typed stream. They can be used within **read:** or **write:** methods for archiving purposes. See the description of **NXReadObject()** in this chapter for more about these methods. Functions are also available for reading and writing certain data types; they're listed below under "SEE ALSO."

These functions are similar to the **NXPrintf()** and **NXScanf()** functions for streams (and to the **printf()** and **scanf()** standard C functions). Before using a typed stream for reading and writing, it must be opened; see the description of **NXOpenTypedStream()** for details on opening a typed stream. (The NXTypedStream type is declared in the header file **objc/typedstream.h**. The structure itself is private since you never need to access its members.)

These four functions take as arguments a pointer to a typed stream, a character string indicating the format of the data to be read or written, and the address of the data. The format string characters and their corresponding data types listed below are supported.

Format Character	**Data Type**
c	char
s	short
i	int
f	float
d	double
@	id
*	char *
%	NXAtom (see text below)
:	SEL
#	class
!	(corresponding data won't be read or written; see below)
{<type>}	struct
[<count><type>]	array

When writing, the "%" format character specifies that data should be written as a **const char** pointer. When reading, the data is read and then converted to a unique string using **NXUniqueString()**. This function is described later in this chapter. The "!" identifier should only be used on data that's the same size as an **int**. The corresponding data item from the stream won't be read or written.

NXReadType() and **NXWriteType()** read and write the data specified by *data* as the single data type specified by *type*. The functions **NXReadTypes()** and **NXWriteTypes()** read and write multiple types of data; the types should be listed in *types* using the appropriate format characters shown above, and matching data should be provided in *data*. This example shows three different data types being written to an already open typed stream:

```
float   aa = 3.0;
int     bb = 5;
char   *cc = "foo";

NXWriteTypes(typedStream, "fi*", &aa, &bb, &cc);
```

If **NXWriteType()** had been used, three lines of code would have been necessary, one for each data type. Both functions take pointers to the data to be written, unlike **printf()**.

To read these three pieces of data from the NXTypedStream, **NXReadTypes()** would be called with the same arguments as shown above for **NXWriteTypes()**:

```
NXReadTypes(typedStream, "fi*", &aa, &bb, &cc);
```

EXCEPTIONS

All four functions check whether the typed stream has been opened for reading or for writing and raise a TYPEDSTREAM_FILE_INCONSISTENCY exception if the type isn't correct. For example, if **NXReadType()** or **NXReadTypes()** is called and the stream was opened for writing, the exception is raised.

The functions for reading raise a TYPEDSTREAM_FILE_INCONSISTENCY exception if the data to be read is not of the expected type.

SEE ALSO

NXOpenTypedStream(), **NXReadObject()**, and **NXReadPoint()**

NXReadTypes() → See NXReadType()

NXReadWordTable(), NXWriteWordTable()

SUMMARY	Read or write Text object's word tables
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <appkit/Text.h>

void **NXReadWordTable**(NXZone **zone*, NXStream **stream*,
    unsigned char ***preSelSmart*, unsigned char ***postSelSmart*,
    unsigned char ***charCategories*, NXFSM ***wrapBreaks*, int **wrapBreaksCount*,
    NXFSM ***clickBreaks*, int **clickBreaksCount*, BOOL **charWrap*)
void **NXWriteWordTable**(NXStream **stream*, const unsigned char **preSelSmart*,
    const unsigned char **postSelSmart*, const unsigned char **charCategories*,
    const NXFSM **wrapBreaks*, int *wrapBreaksCount*, const NXFSM **clickBreaks*,
    int *clickBreaksCount*, BOOL *charWrap*)

DESCRIPTION

These functions read and write the Text object's word tables. Given *stream*, a pointer to a stream containing appropriate data, **NXReadWordTable()** creates word tables in the memory zone specified by *zone*. Conversely, given references to word table structures, **NXWriteWordTables()** records the structures in the stream referred to by *stream*.

The word table arguments taken by these two functions are identical except for the degree of indirection. For each table it will create, **NXReadWordTable()** takes the address of a pointer. When the function returns, these pointers will point to the newly created tables. On the other hand, **NXWriteWordTables()** takes a pointer to each table it will record to the stream.

preSelSmart and *postSelSmart* refer to smart cut and paste tables. These tables specify which characters preceding or following the selection will be treated as equivalent to a space. *wrapBreaks* refers to a break table, the table that a Text object uses to determine word boundaries for line breaks. *wrapBreaksCount* gives the number of elements in the array of NXFSM structures that make up the break table. Similarly, *clickBreaks* and *clickBreaksCount* refer to a click table, the table that determines word boundaries for word selection. Finally, *charWrap* refers to a flag indicating whether words whose length exceeds the Text object's line length should be wrapped on a character-by-character basis.

Word tables can be set through the defaults system. The global parameter NXWordTablesFile determines which word table file an application will use. The value for this parameter can either be a file name or the special values "English" or "C". The special values cause built-in tables for those languages to apply.

EXCEPTIONS

NXReadWordTable() raises an NX_wordTablesRead exception if it's unable to open *stream*. **NXWriteWordTable()** raises an NX_wordTablesWrite exception if it's unable to open *stream* or if *charCategories*, *wrapBreaks*, or *clickBreaks* is NULL.

NXReallyFree() → See NXCreateHashTable()

NXRectClip(), NXRectClipList(), NXRectFill(), NXRectFillList(), NXRectFillListWithGrays(), NXEraseRect(), NXHighlightRect()

SUMMARY Optimize drawing

LIBRARY libNeXT_s.a

SYNOPSIS

#import <appkit/graphics.h>

void **NXRectClip**(const NXRect **aRect*)
void **NXRectClipList**(const NXRect **rects*, int *count*)
void **NXRectFill**(const NXRect **aRect*)
void **NXRectFillList**(const NXRect **rects*, int *count*)
void **NXRectFillListWithGrays**(const NXRect **rects*, const float **grays*, int *count*)
void **NXEraseRect**(const NXRect **aRect*)
void **NXHighlightRect**(const NXRect **aRect*)

DESCRIPTION

These functions provide efficient ways to carry out common drawing operations on rectangular paths.

NXRectClip() intersects the current clipping path with the rectangle referred to by its argument, *aRect*, to determine a new clipping path. **NXRectClipList()** takes an array of *count* number of rectangles and intersects the current clipping path with each of them. Thus, the new clipping path is the graphic intersection of all the rectangles and the original clipping path. Both functions work through the **rectclip** operator. After computing the new clipping path, the current path is reset to empty.

NXRectFill() fills the rectangle referred to by its argument with the current color. **NXRectFillList()** fills a list of *count* rectangles with the current color. Both work through the **rectfill** operator.

NXRectFillListWithGrays() 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*.

As its name suggests, **NXEraseRect()** erases the rectangle referred to by its argument, filling it with white. It does not alter the current color.

NXHighlightRect() 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

NXSetRect(), **NXUnionRect()**

NXRectClipList() → See NXRectClip()

NXRectFill() → See NXRectClip()

NXRectFillList() → See NXRectClip()

NXRectFillListWithGrays() → See NXRectClip()

NXRedComponent(), NXGreenComponent(), NXBlueComponent(), NXCyanComponent(), NXMagentaComponent(), NXYellowComponent(), NXBlackComponent(), NXHueComponent(), NXSaturationComponent(), NXBrightnessComponent(), NXGrayComponent(), NXAlphaComponent()

SUMMARY	Isolate one component of a color
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <appkit/color.h>

float **NXRedComponent(**NXColor *color***)**
float **NXGreenComponent(**NXColor *color***)**
float **NXBlueComponent(**NXColor *color***)**
float **NXCyanComponent(**NXColor *color***)**
float **NXMagentaComponent(**NXColor *color***)**
float **NXYellowComponent(**NXColor *color***)**
float **NXBlackComponent(**NXColor *color***)**
float **NXHueComponent(**NXColor *color***)**
float **NXSaturationComponent(**NXColor *color***)**
float **NXBrightnessComponent(**NXColor *color***)**
float **NXGrayComponent(**NXColor *color***)**
float **NXAlphaComponent(**NXColor *color***)**

DESCRIPTION

Each of these functions takes an NXColor structure as an argument and returns the value of one component of the color, as indicated by the function name.

RETURN

Each functions returns a component of the color passed as an argument. The function name indicates which component is returned. **NXAlphaComponent()** returns NX_NOALPHA if a coverage component is not specified for the color. Otherwise, all return values lie in the range 0.0 through 1.0.

SEE ALSO

NXChangeRedComponent(), **NXSetColor()**, **NXConvertRGBAToColor()**, **NXConvertColorToRGBA()**, **NXEqualColor()**, **NXReadColor()**

NXRegisterDefaults(), NXGetDefaultValue(), NXReadDefault(), NXRemoveDefault(), NXSetDefault(), NXUpdateDefault(), NXUpdateDefaults(), NXWriteDefault(), NXWriteDefaults(), NXSetDefaultsUser()

SUMMARY	Set or read default values
LIBRARY	libdb.a

SYNOPSIS

#import <defaults.h>

int **NXRegisterDefaults**(const char **owner*, const NXDefaultsVector *vector*)
const char ***NXGetDefaultValue**(const char **owner*, const char **name*)
const char ***NXReadDefault**(const char **owner*, const char **name*)
int **NXRemoveDefault**(const char **owner*, const char **name*)
int **NXSetDefault**(const char **owner*, const char **name*, const char **value*)
const char ***NXUpdateDefault**(const char **owner*, const char **name*)
void **NXUpdateDefaults**(void)
int **NXWriteDefault**(const char **owner*, const char **name*, const char **value*)
int **NXWriteDefaults**(const char **owner*, NXDefaultsVector *vector*)
const char ***NXSetDefaultsUser**(const char **newUser*)

DESCRIPTION

Through the defaults system, you can allow users to customize your application to match their preferences by specifying values for default parameters. Each user has a defaults database for storing these default values; it's named **.NeXTdefaults** and resides in **~/.NeXT**.

The defaults registration table allows an application to efficiently read default values for a set of parameters without having to open and close the **.NeXTdefaults** database to obtain each value. The table consists of a list of pairs; each pair is composed of a parameter name and a corresponding default value. The registration table is created at run time by opening the database once to read default values for the parameters the application will use. Every application should create its registration table early in the program, before any default values are needed.

To create this table, call **NXRegisterDefaults()** and give it two arguments: A character string specifying the name of an application, or owner, and an NXDefaultsVector structure. Like the registration table, this structure consists of a list of pairs of parameter names and default values. (It's defined in the header file **defaults.h**.)

The NXDefaultsVector structure serves two purposes. First, it provides a complete list of all parameters that the application will use. Values for all the parameters specified are placed in the registration table at once, so the database doesn't need to be opened and closed for subsequent uses of the parameters. (However, if the application later asks for values for parameters that aren't registered, the database will be opened, read, and closed again.) Second, the structure allows the programmer to suggest values for the parameters. These values are used if the user hasn't stated a preference for a specific value.

If the defaults database doesn't exist when **NXRegisterDefaults()** is called, it's automatically created and placed in the **.NeXT** directory; the directory is also created if necessary.

A good place to call **NXRegisterDefaults()** is in the **initialize** method of the class that will use the parameters. The following example registers the values in **WriteNowDefaults** for the owner **WriteNow**:

```
+ initialize
{
    static NXDefaultsVector WriteNowDefaults = {
        {"NXFont", "Helvetica"},
        {"NXFontSize", "12.0"},
        {NULL}
    };

    NXRegisterDefaults("WriteNow", WriteNowDefaults);

    return self;
}
```

NXRegisterDefaults() creates a registration table that contains a value for each of the parameters listed in the NXDefaultsVector structure. (Note that NULL is used to signal the end of the NXDefaultsVector structure.) This value will be the one listed in the structure if there's no value for that parameter in the database, as described below.

A user's database may contain values for parameters stored multiple times, each with a different owner. For example, the NXFont parameter can have the value Ohlfs with a GLOBAL owner, Times for the owner WriteNow, and Courier for the owner Mail. When searching a user's database for the parameters listed in the NXDefaultsVector structure, **NXRegisterDefaults()** ignores values owned by an application different from the one used as its argument. If it finds a parameter and owner that matches those passed to it as arguments, the corresponding value from the user's database rather than the value from the NXDefaultsVector structure is placed in the registration table. If no parameter-owner match is found, **NXRegisterDefaults()** searches the database's global parameters—that is, those owned by GLOBAL—for a match, and, if it finds one, places the corresponding value in the registration table. If a parameter isn't found in the user's database, the parameter-value pair listed in the NXDefaultsVector structure is placed in the registration table.

Note: When creating their own parameters, applications should use the full market name of their product as the owner of the parameter to avoid colliding with already existing parameters. Noncommercial applications might use the name of the program and the author or institution.

If the application was launched from the command line, any parameter values specified there will be used, overriding values listed in the database and the NXDefaultsVector structure.

To summarize, this is the precedence ordering used to obtain a value for a given parameter for the registration table:

1. The command line
2. The defaults database, with a matching owner
3. The defaults database, with the owner listed as GLOBAL
4. The NXDefaultsVector structure passed to **NXRegisterDefaults()**

When your program needs to use a default value, you'll typically call **NXGetDefaultValue()**. This function takes an owner and name of a parameter as arguments and returns a **char** pointer to the default value for that parameter. **NXRegisterDefaults()** should already have been called, so **NXGetDefaultValue()** first looks in the registration table, where usually it will find a matching parameter and value. If **NXGetDefaultValue()** doesn't find a match in the registration table (which would only be the case if you hadn't listed all parameters when you called **NXRegisterDefaults()**), it searches the **.NeXTdefaults** database for the owner and parameter. If still no match is found, it searches for a matching global parameter, first in the registration table and then in the database. If the value is found in the database rather than the table, **NXRegisterDefaults()** registers that value for subsequent use.

Occasionally, you may want to search only the database for a default value and ignore the command line and the registration table. For example, you might want a value that another application may have changed after the table was created. In these rare cases call **NXReadDefault()**, which takes an owner and the parameter as arguments and looks in the database for an exact match. It doesn't look for a global parameter unless GLOBAL is specified as the owner. If a match is found, a **char** pointer to the default value is returned; if no value is found, NULL is returned. After obtaining a value from the database with **NXReadDefault()**, you may want to write it into the registration table with **NXSetDefault()**.

NXSetDefault() takes as arguments an owner, the name of a parameter, and a value for that parameter. The parameter and its default value are placed in the registration table, but they aren't written into the **.NeXTdefaults** database.

NXRemoveDefault() removes the specified default value from the database.

NXWriteDefault() writes the value and default parameter specified as its arguments into the database and places them in the registration table. Similarly, **NXWriteDefaults()** writes a vector of defaults into the database and registers it. Both **NXWriteDefault()** and **NXWriteDefaults()** return the number of successfully written values. To maximize efficiency, you should use one call to **NXWriteDefaults()** rather than several calls to **NXWriteDefault()** to write multiple values. This will save the time required to open and close the database each time a value is written.

Since other applications (and the user) can write to the database, at various points the database and the registration table might not agree on the value of a given parameter. You can update the registration table with any changes that have been made to the database since the table was created by calling **NXUpdateDefault()** or **NXUpdateDefaults()**. Both functions compare the table and the database. If a value is found in the database that is newer than the corresponding value in the registration table, the new value is written into the registration table.

NXUpdateDefault() updates the value for the single parameter and owner given as its arguments. **NXUpdateDefaults()**, which takes no arguments, updates the entire registration table. It checks every parameter in the registration table, determines whether a newer value exists in the database, and puts any newer values it finds in the registration table.

Ordinarily, the defaults database functions access the database belonging to the user who started the application. **NXSetDefaultsUser()** changes the defaults database accessed by subsequent calls to these functions. **NXSetDefaultsUser()** accepts the name of a user whose database you wish to access; it returns a pointer to the name of the user whose defaults database was previously set for access by these functions. All entries in the registration table are purged; use **NXGetDefaultValue()** or **NXRegisterDefaults()** to get the new user's defaults for your application. When **NXSetDefaultsUsers()** is called, the user who started the application must have appropriate access (read, write, or both) to the defaults database of the new user. This function is generally called in applications intended for use by a superuser who needs to update defaults databases for a number of users.

RETURN

NXRegisterDefaults() returns 0 if the database couldn't be opened; otherwise it returns 1.

NXGetDefaultValue() returns a **char** pointer to the requested default value or 0 if the database couldn't be opened.

NXReadDefault() returns a **char** pointer to the default value; if a value is not found, NULL is returned.

NXRemoveDefault() returns 1 or 0 if the default couldn't be removed.

NXSetDefault() returns 1 if it successfully set a default value and 0 if not.

NXUpdateDefault() returns the new value or NULL if the value did not need to be updated.

NXWriteDefault() returns 1 unless an error occurs while writing the default, in which case it returns 0.

NXWriteDefaults() returns the number of successfully written default values.

NXSetDefaultsUser() returns the login name of the user whose defaults database was being accessed before the function was called.

NXRegisterErrorReporter(), NXRemoveErrorReporter(), NXReportError()

SUMMARY	Define an error reporter
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <appkit/errors.h>

void **NXRegisterErrorReporter**(int *min*, int *max*,
    void (**proc*)(NXHandler **errorState*))
void **NXRemoveErrorReporter**(int *code*)
void **NXReportError**(NXHandler **errorState*)

DESCRIPTION

These three functions set up an error reporting procedure, which typically includes writing a message to **stderr**. When an error is raised (using **NX_RAISE**()), each of the nested error handlers are notified successively until one can handle the error without forwarding it to the next level. This handler executes its error handling code, which usually includes calling **NXReportError**().

NXReportError()'s *errorState* argument contains information about the error, including an error code that identifies the error. (The NXHandler structure is defined in the header file **streams/error.h**.) **NXReportError**() uses this error code to search the codes for which error reporters have been registered (see below). When it finds a match, it calls the corresponding procedure. If no matching error code is found, an unknown error code message is written to **stderr**.

The Application Kit registers its error reporters in the **initialize** class method of the Application object. Other applications that subclass Application will use these reporters by default, but they can also define their own set of errors and a reporter. To create your own range of error codes and corresponding error messages, call **NXRegisterErrorReporter**(). Its first two arguments define the range of numbers you will use as error codes. Applications that are defining their own reporter should begin their range at NX_APPBASE. The third argument points to the procedure that matches an error code in that range with an error message.

NXRemoveErrorReporter() removes the error reporter that had been assigned to the error *code* passed in as its argument.

SEE ALSO

NX_RAISE(), **NXDefaultTopLevelErrorHandler()**

NXRegisterPrintfProc()

SUMMARY	Register a procedure for formatting data written to a stream
LIBRARY	libsys_s.a

SYNOPSIS

#import <streams/streams.h>

void **NXRegisterPrintfProc(**char *formatChar*, NXPrintfProc **proc*, void **procData***)**

DESCRIPTION

NXRegisterPrintfProc registers *formatChar*, a format character that corresponds to **proc*, which is a pointer to a function of type NXPrintfProc. The type definition for an NXPrintfProc function is:

```
typedef   void   NXPrintfProc(NXStream *stream, void *item,
                              void *prodCata)
```

formatChar can be any of the characters "vVwWyYzZ"; other characters are reserved for use by NeXT. *procData* represents client data that will be blindly passed along to the function.

After calling **NXRegisterPrintfProc()**, *formatChar* can be used in a format string for the **NXPrintf()** or **NXVPrintf()** functions. When these functions encounter *formatChar* in a format string, *proc* will be called to format the corresponding argument passed to **NXPrintf()**. For example:

```
tabOver(NXStream stream, void *item, void *data)
{
...
}

NXRegisterPrintfProc('v', &tabOver, NULL)
...
NXPrintf(myStream, "%v", itemOne)
```

This code registers "v" as the formatting character for tabOver(); with the NULL argument, no client data will be passed to the tabOver() function. **NXPrintf()** then passes the variable itemOne to tabOver for formatting, which formats the item and places it in myStream.

SEE ALSO

NXPutc()

NXRemoteMethodFromSel(), NXResponsibleDelegate()

SUMMARY	Match an Objective-C method and a receiver to a remote message
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <appkit/ Listener.h>

NXRemoteMethod ***NXRemoteMethodFromSel(**SEL *aSelector*,
  NXRemoteMethod **methods*)
id **NXResponsibleDelegate(**Listener **aListener*, SEL *aSelector*)

DESCRIPTION

These two functions are used within subclasses of the Listener class. When you define a Listener subclass using the **msgwrap** utility, calls to these functions are generated automatically.

NXRemoteMethodFromSel() looks up the *aSelector* method in a table of remote methods that have been declared for the Listener subclass. The second argument, *methods*, is a pointer to the beginning of the table. A pointer to the table entry for the *aSelector* method is returned.

NXResponsibleDelegate() returns the **id** of the object that responds to *aSelector* remote messages received by *aListener*. That object will be the Listener's delegate, or the delegate of the Listener's delegate. A Listener normally entrusts the remote messages it receives to its delegate, but if its delegate has a delegate of its own, the Listener defers to that object. Thus if the Application object is the Listener's delegate, the Application object's delegate will be given the first chance to respond to *aSelector* messages.

RETURN

NXRemoteMethodFromSel() returns a pointer to the entry for the *aSelector* method in a table of remote methods kept by a Listener subclass, or NULL if there is no entry for the method.

NXResponsibleDelegate() returns the delegate that responds to *aSelector* remote messages received by *aListener.* If the delegate of *aListener*'s delegate can respond to *aSelector* messages, it is returned. If not and *aListener*'s delegate can respond to *aSelector* messages, it is returned. If neither delegate responds to *aSelector* messages (or *aListener* doesn't have a delegate), **nil** is returned.

NXRemoveDefault() → See NXRegisterDefaults()

NXRemoveErrorReporter() → See NXRegisterErrorReporter()

NXReportError() → See NXRegisterErrorReporter()

NXResetErrorData() → See NXAllocErrorData()

NXResetHashTable() → See NXCreateHashTable()

NXResetUserAbort() → See NXUserAbort()

NXResponsibleDelegate() → See NXRemoteMethodFromSel()

NXRunAlertPanel(), NXGetAlertPanel(), NXFreeAlertPanel()

SUMMARY	Create or free an attention panel
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <appkit/Panel.h>

int **NXRunAlertPanel**(const char **title*, const char **msg*, const char **defaultButton*,
  const char **alternateButton*, const char **otherButton*, ...)
id **NXGetAlertPanel**(const char **title*, const char **msg*, const char **firstButton*,
  const char **alternateButton*, const char **otherButton*, ...)
void **NXFreeAlertPanel**(id *alertPanel*)

DESCRIPTION

NXRunAlertPanel() and **NXGetAlertPanel**() both create 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. **NXRunAlertPanel**() creates the panel and runs it in a modal event loop; **NXGetAlertPanel**() returns the **id** of a panel that you can use in a modal session.

These functions take the same set of arguments. 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 UNIX manual 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 **NXRunAlertPanel**() must have at least one button, which will have the symbol for the Return key; if you pass a NULL title to the other two buttons, they won't be created. If NULL is passed as the *defaultButton*, "OK" will be used as its title. The panel created by **NXGetAlertPanel**() doesn't have to have any buttons. If you supply a title for *firstButton*, it will be displayed with the symbol for the Return key.

NXRunAlertPanel() 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. **NXRunAlertPanel**() then removes the panel from the screen list and returns a value that indicates which of the three buttons the user clicked: NX_ALERTDEFAULT, NX_ALERTALTERNATE, or NX_ALERTOTHER. (If an error occurred while creating the panel, NX_ALERTERROR is returned.) For efficiency, **NXRunAlertPanel**() creates the panel the first time it's called and reuses it on subsequent calls, reconfiguring it if necessary.

NXGetAlertPanel() doesn't set up a modal event loop; instead, it returns the **id** of a panel that can be used to set up a modal session. A modal sessions 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.

To set up a modal session, send the Application object a **beginModalSession:for:** message with the **id** returned by **NXGetAlertPanel**() 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 **NXGetAlertPanel**(), you must free it by calling **NXFreeAlertPanel**(). This function takes the **id** returned by **NXGetAlertPanel**() as its only argument.

RETURN

NXRunAlertPanel() returns a constant that indicates which button in the attention panel the user clicked.

NXGetAlertPanel() returns the **id** of an attention panel for use in a modal session.

NXSaturationComponent() → See NXRedComponent()

NXSaveToFile() → See NXOpenMemory()

NXScanALine(), NXDrawALine()

SUMMARY	Calculate or draw line of text (in Text object)
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <appkit/Text.h>

int **NXScanALine**(id *self*, NXLayInfo **layInfo*)
int **NXDrawALine**(id *self*, NXLayInfo **layInfo*)

DESCRIPTION

A Text object calls the first two functions to calculate and draw a line of text. Each function's first argument is a reference to the Text object's **id**. The second argument is an NXLayInfo structure, which is defined in the header file **appkit/Text.h**.

To determine the placement of characters in a line, **NXScanALine()** takes into account line width, text alignment, font metrics, and other data from the Text object. It stores the results of its calculations in global variables.

A Text object calls **NXDrawALine()** to draw a line of text. The global variables set by **NXScanALine()** provide **NXDrawALine()** with the information it needs to draw each line of text.

RETURN

NXScanALine() returns 1 only if a word's length exceeds the width of a line and the Text object's **charWrap** instance variable is NO. Otherwise, it returns 0.

NXDrawALine() has no significant return value.

NXScanf() → See NXPutc()

NXSeek(), NXTell(), NXAtEOS()

SUMMARY Set or report current position in a stream

LIBRARY libsys_s.a

SYNOPSIS

#import <streams/streams.h>

void **NXSeek**(NXStream **stream*, long *offset*, int *ptrName*)
long **NXTell**(NXStream **stream*)
BOOL **NXAtEOS**(NXStream **stream*)

DESCRIPTION

These functions set or report the current position in the stream given as an argument. This position determines which data will be read next or where the next data will be written since the functions for reading and writing to a stream start from the current position.

NXSeek() sets the position *offset* number of bytes from the place indicated by *ptrName*, which can be NX_FROMSTART, NX_FROMCURRENT, or NX_FROMEND.

NXTell() returns the current position of the buffer. This information can then be used in a call to **NXSeek()**.

The macro **NXAtEOS()** evaluates to TRUE if the end of a stream has been reached. Since streams opened for writing don't have an end, this macro should only be used with streams opened for reading.

Since position within a Mach port stream is undefined, **NXSeek()** and **NXTell()** shouldn't be called on a Mach port stream. These functions also shouldn't be used on a typed stream. The NX_CANSEEK flag (defined in the header file **streams/streams.h**) can be used to determine if a given stream is seekable.

RETURN

NXTell() returns the current position of the buffer.

NXAtEOS() evaluates to TRUE if the end of the stream has been detected and to FALSE otherwise.

EXCEPTIONS

NXSeek() and **NXTell()** raise an NX_illegalStream exception if the stream passed in is invalid.

NXSeek() raises an NX_illegalSeek exception if *offset* is less than 0 or greater than the length of a reading stream. This exception will also be raised if *ptrName* is anything other than the three constants listed above.

NXSetColor()

SUMMARY Set the current color

LIBRARY libNeXT_s.a

SYNOPSIS

#import <appkit/color.h>

void **NXSetColor**(NXColor *color*)

DESCRIPTION

This function uses PostScript operators to make *color* the current color of the current graphics state. If *color* includes a coverage component (if **NXAlphaComponent()** returns anything but NX_NOALPHA), it also sets the current coverage. However, coverage will not be set when printing.

SEE ALSO

NXEqualColor(), **NXConvertRGBAToColor()**, **NXConvertColorToRGBA()**, **NXRedComponent()**, **NXChangeRedComponent()**, **NXReadColor()**

NXSetDefault() → See NXRegisterDefaults()

NXSetDefaultsUser() → See NXRegisterDefaults()

NXSetExceptionRaiser() → See NXDefaultExceptionRaiser()

NXSetGState(), NXCopyCurrentGState()

SUMMARY Set or copy current graphics state object

LIBRARY libNeXT_s.a

SYNOPSIS

#import <appkit/publicWraps.h>

void **NXSetGState**(int *gstate*)
void **NXCopyCurrentGState**(int *gstate*)

DESCRIPTION

These functions set the current PostScript graphics state.

NXSetGState() is a C function cover for the PostScript **setgstate** operator. It sets the current graphics state to that specified by *gstate*.

NXCopyCurrentGState() takes a snapshot of the current graphic state and assigns it the number *gstate*. Generally, a snapshot should be taken only when the current path is empty and the current clip path is in its default state.

NXSetRect(), NXOffsetRect(), NXInsetRect(), NXIntegralRect(), NXDivideRect()

SUMMARY Modify a rectangle

LIBRARY libNeXT_s.a

SYNOPSIS

#import <appkit/graphics.h>

void **NXSetRect**(NXRect **aRect*, NXCoord *x*, NXCoord *y*, NXCoord *width*,
  NXCoord *height*)
void **NXOffsetRect**(NXRect **aRect*, NXCoord *dx*, NXCoord *dy*)
void **NXInsetRect**(NXRect **aRect*, NXCoord *dx*, NXCoord *dy*)
void **NXIntegralRect**(NXRect **aRect*)
NXRect ***NXDivideRect**(NXRect **aRect*, NXRect **bRect*, NXCoord *slice*, int *edge*)

DESCRIPTION

These functions modify the *aRect* argument. It's assumed that all arguments are expressed within the same coordinate system.

The first function, **NXSetRect**(), sets the values in the NXRect structure specified by its first argument, *aRect*, to the values passed in the other arguments. It provides a convenient way to initialize an NXRect structure.

The next two functions, **NXOffsetRect**() and **NXInsetRect**(), are illustrated in Figure 3-3.

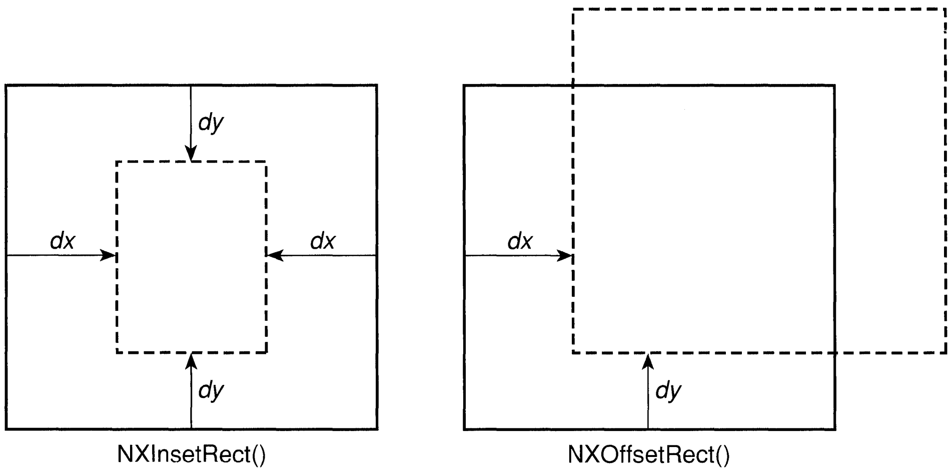


Figure 3-3. Inset and Offset Rectangles

NXOffsetRect() shifts the location of the rectangle by *dx* along the x-axis and by *dy* along the y-axis. **NXInsetRect()** alters the rectangle so that the two sides that are parallel to the y-axis are inset by *dx* and the two sides parallel to the x-axis are inset by *dy.*

NXIntegralRect() alters the rectangle so that none of its four defining values (*x*, *y*, *width*, and *height*) have fractional parts. The values are raised or lowered to the nearest integer, as appropriate, so that the new rectangle completely encloses the old rectangle. These alterations ensure that the sides of the new rectangle lie on pixel boundaries, if the rectangle is defined in a coordinate system that has its coordinate origin on the corner of four pixels and a unit of length along either axis equal to one pixel. If the rectangle's width or height is 0 (or negative), it's set to a rectangle with origin at (0.0, 0.0) and with 0 width and height.

NXDivideRect() divides a rectangle in two. It cuts a slice off the rectangle specified by *aRect* to form a new rectangle, which it stores in the structure specified by *bRect*. The rectangle specified by *aRect* is modified accordingly. The size of the slice taken from the rectangle is indicated by *slice*; it's taken from the side of the rectangle indicated by *edge*. The values for *edge* can be:

0	The slice is made parallel to the y-axis, along the side with the smallest x coordinate values.
1	The slice is made parallel to the x-axis, along the side with the smallest y coordinate values.
2	The slice is made parallel to the y-axis, along the side with the greatest x coordinate values.
3	The slice is made parallel to the x-axis, along the side with the greatest y coordinate values.

RETURN

NXSetRect(), **NXOffsetRect()**, **NXInsetRect()**, and **NXIntegralRect()** have no significant return values. **NXDivideRect()** returns a pointer to the new rectangle, *bRect*.

SEE ALSO

NXUnionRect(), **NXMouseInRect()**

NXSetServicesMenuItemEnabled(), NXIsServicesMenuItemEnabled()

SUMMARY	Determine whether an item is included in Services menus
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <appkit/Listener.h >

int **NXSetServicesMenuItemEnabled**(const char **item*, BOOL *flag*)
BOOL **NXIsServicesMenuItemEnabled**(const char **item*)

DESCRIPTION

NXSetServicesMenuItemEnabled() is used by a service-providing application to determine whether the Services menus of other applications will contain the *item* command enabling users to request its services. If *flag* is YES, the Application Kit will build Services menus for other applications that include the *item* command. If *flag* is NO, *item* won't appear in any application's Services menu. *item* should be the same character string entered in the "Menu Item:" field of the __services section. All service providers are required to have this section.

Service-providing applications should let users decide whether the Services menus of other applications they use should include the *item* command.

RETURN

NXSetServicesMenuItemEnabled() returns 0 if it's successful in enabling or disabling the *item* command, and a number other than 0 if not. **NXIsServicesMenuItem()** returns YES if *item* is currently enabled, and NO if it's not.

NXSetTopLevelErrorHandler() → See NXDefaultTopLevelErrorHandler()

NXSetTypedStreamZone() → See NXGetTypedStreamZone()

NXSetUncaughtExceptionHandler(), NXGetUncaughtExceptionHandler()

SUMMARY	Handle uncaught exceptions
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <objc/error.h >

void **NXSetUncaughtExceptionHandler**(NXUncaughtExceptionHandler **proc*)
NXUncaughtExceptionHandler ***NXGetUncaughtExceptionHandler**(void)

DESCRIPTION

These macros provides a means of handling exceptions that are raised outside of an NX_DURING...NX_ENDHANDLER construct. You can use the Application object's default procedure, or you can define your own handler using **NXSetUncaughtExceptionHandler**().

If *proc* is NULL or if you never call **NXSetUncaughtExceptionHandler**(), your program will use the Application object's default procedure. This function writes an uncaught exception message to **stderr** if the application was launched from a terminal. If the application was launched by the Workspace Manager, the message is written using **syslog**() with the priority set to LOG_ERR; this message will normally appear in the Workspace Manager's console window. The default uncaught exception handler then calls the function pointed to by **NXTopLevelErrorHandler**() and passes it any data about the exception supplied by **NX_RAISE**(), which was called when the exception occurred. (See the description of **NX_RAISE**().) If you haven't defined your own top-level error handler, the program exits.

To create your own handler, you define an exception handling function and give the name of that function as an argument to **NXSetUncaughtExceptionHandler**(). Subsequent calls to **NXGetUncaughtExceptionHandler**() will return a pointer to the function. These two macros are defined in the header file **streams/error.h**.

SEE ALSO

NX_RAISE(), **NXDefaultTopLevelErrorHandler**()

NXSizeBitmap() → See NXImageBitmap()

NXStreamCreateFromZone(), NXStreamCreate(), NXStreamDestroy(), NXDefaultRead(), NXDefaultWrite(), NXFill(), NXChangeBuffer()

SUMMARY	Support a user-defined stream
LIBRARY	libsys_s.a

SYNOPSIS

#import <streams/streamsimpl.h>

NXStream ***NXStreamCreateFromZone**(int *mode*, int *createBuf*, NXZone **zone*)
NXStream ***NXStreamCreate**(int *mode*, int *createBuf*)
void **NXStreamDestroy**(NXStream **stream*)
int **NXDefaultRead**(NXStream **stream*, void **buf*, int *count*)
int **NXDefaultWrite**(NXStream **stream*, const void **buf*, int *count*)
int **NXFill**(NXStream **stream*)
void **NXChangeBuffer**(NXStream **stream*)

DESCRIPTION

These functions need only be used if you implement your own version of a stream. If you're using a memory stream, a stream on a file, a stream on a Mach port, or a typed stream, you don't need the functions described here. Instead, you can just use the functions already defined for these types of streams; see the *Technical Summaries* manual for a list of these functions.

The first argument to **NXStreamCreateFromZone**(), *mode*, indicates whether the stream to be created will be used for reading or writing or both. It should be one of the following constants: NX_READONLY, NX_WRITEONLY, or NX_READWRITE. The argument *createBuf* specifies whether the stream should be buffered. If it is TRUE, a buffer is created of size NX_DEFAULTBUFSIZE, as defined in the header file **streams/streamsimpl.h**. The argument *zone* specifies the memory zone where you allocate memory for the new stream; see **NXZoneMalloc()** for more on allocating zones of memory. When implementing your own version of a stream, you may want to provide a function to open such a stream; this function will probably call **NXStreamCreateFromZone**(), as **NXOpenMemory**(), **NXOpenPort**(), and **NXOpenFile**() do.

NXStreamCreate() calls **NXStreamCreateFromZone**() with the default zone as its *zone* argument.

NXStreamDestroy() destroys the stream given as its argument, deallocating the space it had used. If a buffer had been created for *stream*, its storage is also freed. To avoid losing data, a stream should be flushed using **NXFlush**() before it's destroyed. When implementing your own version of a stream, you may want to provide a function to close such a stream; this function will probably call **NXStreamDestroy**(), as **NXClose**()and **NXCloseMemory**() do.

NXDefaultRead() and **NXDefaultWrite**() read and write multiple bytes of data on a stream. **NXDefaultRead**() reads the next *count* number of bytes from *stream*, starting at the position specified by the buffer pointer *buf*. **NXDefaultWrite**() writes *count* number of bytes to *stream*, starting at the position specified by *buf*. These functions return the number of bytes read or written. When implementing your own version of a stream, you can use these functions with your stream unless you want to perform specialized buffer management. If you implement your own versions of these functions for reading and writing bytes, they should return the number of bytes read or written.

When reading from a buffered stream, **NXFill**() can be called to fill the buffer with the next data to be read. Check whether **buf_left** is equal to 0 to determine whether all the data currently in the buffer has been read. (See the header file **streams/streams.h** for more information about **buf_left**, which is part of an NXStream structure.)

NXChangeBuffer() switches the mode of a stream between reading and writing. If the argument *stream* had been defined for reading, this function changes it to a stream that can be written to; if *stream* had been defined for writing, it becomes a stream for reading. In both cases, the pointer that points to either the next piece of data to be read from the buffer or the next location to which data will be written is realigned appropriately. Also, NX_READFLAG and NX_WRITEFLAG are updated to reflect the new mode of the stream.

RETURN

NXStreamCreate() returns a pointer to the stream it creates.

NXDefaultRead() and **NXDefaultWrite**() return the number of bytes read or written.

NXFill() returns the number of characters read into the buffer.

EXCEPTIONS

All functions that take a stream as an argument raise an NX_illegalStream exception if the stream passed in is invalid.

NXFill() raises an NX_illegalRead exception if an error occurs while filling.

NXChangeBuffer() raises an NX_illegalStream exception if NX_READFLAG and NX_WRITEFLAG have not been set to match the NX_CANREAD and NX_CANWRITE flags.

SEE ALSO

NXOpenFile(), **NXOpenMemory**(), **NXClose**(), **NXFlush**(), **NXRead**()

NXStreamDestroy() → See NXStreamCreate()

NXStrHash() → See NXCreateHashTable()

NXStrIsEqual() → See NXCreateHashTable()

NXSystemVersion()

SUMMARY Return the system version for reading streams

LIBRARY libsys_s.a

SYNOPSIS

#import <objc/typedstreams.h>

int **NXSystemVersion**(NXTypedStream **stream*)

DESCRIPTION

NXSystemVersion returns the NeXT system version used for writing *stream*. The system version is useful if the methods or data types defined for the class of the object archived in *stream* have changed from one version to another, by enabling you to test the version and switch code to handle the object depending on the version. This function is only useful with streams opened for reading.

RETURN

This function returns an integer value corresponding to one of the system version constants listed in Chapter 1, "Constants and Data Types."

NXTell() → See NXSeek()

NXTextFontInfo()

SUMMARY Calculate font ascender, descender, and line height

LIBRARY libNeXT_s.a

SYNOPSIS

#import <appkit/Text.h>

void **NXTextFontInfo**(id *fontId*, NXCoord **ascender*, NXCoord **descender*, NXCoord **lineHeight*)

DESCRIPTION

Given a Font object's **id**, **NXTextFontInfo()** calculates the ascender, descender, and line height values for that font. *fontId* is the Font object's **id**. *ascender*, *descender*, and *lineHeight* are the addresses that will hold the ascender, descender, and line height values after a call to **NXTextFontInfo()**.

NXToAscii(), NXToLower(), NXToUpper()

SUMMARY	Convert NeXTstep–encoded characters
LIBRARY	libsys_s.a

SYNOPSIS

#import <NXCType.h>

unsigned char ***NXToAscii**(unsigned *c*)
int **NXToLower**(unsigned *c*)
int **NXToUpper**(unsigned *c*)

DESCRIPTION

These functions convert characters encoded in the extended character set defined by NeXTstep encoding. They are similar to the standard C library functions **toascii**(), **tolower**(), and **toupper**() (see the UNIX manual page for ctype), which operate on characters in the ASCII character set.

NXToLower() converts an upper–case letter to its lower–case equivalent, and **NXToUpper()** converts a lower–case letter to its upper–case equivalent. If there's no opposite case equivalent—or if the character is already of the desired case—these functions return the supplied argument unchanged.

NXToAscii() converts its argument to a value that lies within the standard ASCII character set. The lower 128 positions in the NeXTstep encoding constitute the ASCII character set, so no conversion is required for codes in this range. For the upper 128 character codes—the extended characters—**NXToAscii()** makes these conversions:

Extended Character	Converts to
Agrave, Aacute, Acircumflex, Atilde, Adieresis, Aring	A
Ccedilla	C
Egrave, Eacute, Ecircumflex, Edieresis	E
Igrave, Iacute, Icircumflex, Idieresis	I
Ntilde	N
Ograve, Oacute, Ocircumflex, Otilde, Odieresis, Oslash	O
Ugrave, Uacute, Ucircumflex, Udieresis	U
Yacute	Y
eth, Eth	TH
Thorn, thorn	th
fi	fi
fl	fl
agrave, aacute, acircumflex, atilde, adieresis, aring	a
ccedilla	c
egrave, eacute, ecircumflex, edieresis	e
AE	AE
igrave, iacute, icircumflex, idieresis	i
ntilde	n
Lslash	L
OE	OE
ograve, oacute, ocircumflex, otilde, odieresis, oslash	o
ae	ae
ugrave, uacute, ucircumflex, udieresis	u
dotlessi	i
yacute, ydieresis	y
lslash	l
oe	oe
germandbls	ss
multiply	x
divide	/
exclamdown	!
quotesingle	'
quotedblleft, guillemotleft, quotedblright, guillemotright, quotedblbase	\
quotesinglbase	'
guilsinglleft	<
guilsinglright	>
periodcentered	.
brokenbar	|
bullet	*
ellipsis	...
questiondown	?
onesuperior	1
twosuperior	2
threesuperior	3
emdash	-
plusminus	+-
onequarter	1/4

(continued)

Extended Character	**Converts to**
onehalf	1/2
threequarters	3/4
ordfeminine	a
ordmasculine	o
mu, copyright, cent, sterling, fraction, yen, florin, section, currency, registered, endash, dagger, daggerdbl, paragraph, perthousand, logicalnot, grave, acute, circumflex, tilde, macron, breve, dotaccent, dieresis, ring, cedilla, hungarumlaut, ogonek, caron,	_

RETURN

NXToAscii() returns by reference a valid ASCII character. **NXToLower**() or **NXToUpper**() returns an integer value that represents the converted character.

SEE ALSO

NXIsAlpha()

NXToLower() → See NXToAscii()

NXTopLevelErrorHandler() → See NXDefaultTopLevelErrorHandler()

NXToUpper() → See NXToAscii()

NXTypedStreamClassVersion()

SUMMARY Get the class version number of an archived instance

LIBRARY libsys_s.a

SYNOPSIS

#import <objc/typedstream.h>

int **NXTypedStreamClassVersion**(NXTypedStream **typedStream*,
    const char **className*)

DESCRIPTION

This function returns the class version number of an archived object. Class versioning is useful if you create a class, archive an instance of it, then change the class—by adding instance variables to it, for example. This function is used in a class's **read:** method to select the appropriate code for initializing the instance being unarchived. This function should be called only on a typed stream opened for reading with **NXReadObject**().

NXTypedStreamClassVersion() can be called in your **read:** method after sending a **[super read:***typedStream*] message and before performing version-specific initialization. Calling this function doesn't change the position of the read pointer in *typedStream*. If you need to know the version of an object's superclass (or any class in its inheritence hierarchy), call this function using the name of that class as *className*.

For **NXTypedStreamClassVersion**() to return a non-zero value, you should change the class version to a new value whenever you change the class definition. The Object class provides two methods for handling class versioning. Object's **setVersion:** class method can be used in a subclass's **initialize** class method to set a new class version when you change the instance variables. Object's **version** class method returns the current version of your class.

The **NXWriteObject**() function automatically archives the class version when it is archiving an object. The default version number is 0. Thus if you have previously archived instances of a class without setting the version, you can set the version of the altered class to any integer value other than 0, then use this function to detect old and new instances of the class.

In the following code example, MyClass's **initialize** method sets the class version using Object's **setVersion:** method:

```
@implementation MyClass:MySuperClass
+ initialize
{
    [MyClass setVersion:MYCLASS_CURRENT_VERSION];
    return self;
}
```

In the next example, MyClass's **read:** method uses version numbers to unarchive old and new instances differently:

```
- read:(NXTypedStream *)typedStream
{
    [super read:typedStream];
    if (NXTypedStreamClassVersion(typedStream, "MyClass") ==
        [MyClass version] {
        /* read code for current version */
            . . .
    }
    else {
        /* read code for old version */
        . . .
    }
}
```

See the description of **NXReadObject()** earlier in this chapter for more information about archiving. The NXTypedStream type is declared in the header file **objc/typedstream.h**. The structure itself is private since you never need to access its members.

SEE ALSO

NXReadObject()

NXUngetc() → See NXPutc()

NXUnionRect(), NXIntersectionRect()

SUMMARY	Compute third rectangle from two rectangles
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <appkit/graphics.h>

NXRect ***NXUnionRect**(const NXRect **aRect*, NXRect **bRect*)
NXRect ***NXIntersectionRect**(const NXRect **aRect*, NXRect **bRect*)

DESCRIPTION

NXUnionRect() figures the graphic union of two rectangles—that is, the smallest rectangle that completely encloses both. It takes pointers to the two rectangles as arguments and replaces the second rectangle with their union. If one rectangle has zero (or negative) width or height, *bRect* is replaced with the other rectangle. If both of the rectangles have 0 (or negative) width or height, *bRect* is set to a rectangle with its origin at (0.0, 0.0) and with 0 width and height.

NXIntersectionRect() figures the graphic intersection of two rectangles—that is, the smallest rectangle enclosing any area they both have in common. It takes pointers to the two rectangles as arguments. If the rectangles overlap, it replaces the second one, *bRect*, with their intersection. If the two rectangles don't overlap, *bRect* is set to a rectangle with its origin at (0.0, 0.0) and with a 0 width and height. Adjacent rectangles that share only a side are not considered to overlap.

Both functions assume that all arguments are expressed within the same coordinate system.

RETURN

NXUnionRect() returns its second argument (*bRect*), a pointer to the union of the two rectangles unless both rectangles have 0 (or negative) width or height, in which case it returns a pointer to a NULL rectangle.

If the two rectangles overlap, **NXIntersectionRect**() returns its second argument (*bRect*), a pointer to their intersection. If the rectangles don't overlap, it returns a pointer to a NULL rectangle.

SEE ALSO

NXIntersectsRect()

NXUniqueString(), NXUniqueStringWithLength(), NXUniqueStringNoCopy(), NXCopyStringBuffer(), NXCopyStringBufferFromZone()

SUMMARY	Manipulate a string buffer
LIBRARY	libsys_s.a

SYNOPSIS

#import <objc/hashtable.h >

NXAtom **NXUniqueString**(const char **buffer*)
NXAtom **NXUniqueStringWithLength**(const char **buffer*, int *length*)
NXAtom **NXUniqueStringNoCopy**(const char **buffer*)
char ***NXCopyStringBuffer**(const char **buffer*)
char ***NXCopyStringBufferFromZone**(const char **buffer*, NXZone **zone*)

DESCRIPTION

The first three functions in this group create unique strings, which are allocated once and then can be shared. The fourth and fifth function allocates memory for and returns a copy of the given string.

Unique strings are identified by the type NXAtom, which indicates that they can be compared using == rather than **strcmp()**. NXAtom strings shouldn't be deallocated or modified; the Mach function **vm_protect()** is used to ensure that the strings are read-only. (The type NXAtom is defined in **objc/hashtable.h**.)

NXUniqueString(), **NXUniqueStringWithLength()**, and **NXUniqueStringNoCopy()** maintain a hash table of unique strings. Each function checks if the string passed in is already in the table and if so, returns it. Because a hash table is used, the average search time is constant regardless of how many unique strings exist. If *buffer* doesn't exist in the hash table, **NXUniqueString()** and **NXUniqueStringWithLength()** return a pointer to a copy of it as an NXAtom; **NXUniqueStringNoCopy()** inserts the string in the hash table but doesn't make a copy of it. For efficiency, all unique strings are stored in the same area of virtual memory.

NXUniqueString() assumes *buffer* is null-terminated; if it's NULL, **NXUniqueString()** returns NULL. **NXUniqueStringWithLength()** assumes that *buffer* is a non-NULL string of at least *length* non-NULL characters.

NXCopyStringBuffer() allocates memory from the default memory zone for a copy of *buffer*. Then *buffer*, which should be null-terminated, is copied using **strcpy()**. **NXCopyStringBufferFromZone()** is identical to **NXCopyStringBuffer()** except that memory is allocated from the specified zone.

RETURN

NXUniqueString() and **NXUniqueStringWithLength()** return a pointer to a copy of *buffer* as an NXAtom.

NXUniqueStringNoCopy() returns a pointer to the string passed in.

NXCopyStringBuffer() and **NXCopyStringBufferFromZone()** return a pointer to a copy of *buffer*.

NXUniqueStringNoCopy() → See NXUniqueString()

NXUniqueStringWithLength() → See NXUniqueString()

NXUnnameObject() → See NXGetNamedObject()

NXUpdateDefault() → See NXRegisterDefaults()

NXUpdateDefaults() → See NXRegisterDefaults()

NXUpdateDynamicServices()

SUMMARY Re-register provided services

LIBRARY libNeXT_s.a

SYNOPSIS

#import <appkit/Listener.h>

void **NXUpdateDynamicServices**(void)

DESCRIPTION

NXUpdateDynamicServices() is used by a service-providing application to re-register the services it is willing to provide. A list of an application's dynamic services should be maintained in the user's **~/.NeXT/services** directory; this list is syntactically identical to the list in the application's __services section. Thus, an application named Foo should maintain its dynamic services in the **~/.NeXT/services/Foo** file. Many applications do not provide dynamic services; all the services they provide are known at compile time, so their services are simply listed in their __services section. If the services an application can provide may change at run time, the application can build a list of additional services that it is willing to provide and then call **NXUpdateDynamicServices()** to make these services available. An example of a dynamic service provider is Digital Librarian™; when you drag a folder named "Business" into its Librarian Services window, the Digital Librarian will update its services in order to provide a "Search in Business" service.

NXUserAborted(), NXResetUserAbort()

SUMMARY Report user's request to abort

LIBRARY libNeXT_s.a

SYNOPSIS

#import <appkit/Application.h>

BOOL **NXUserAborted**(void)
void **NXResetUserAbort**(void)

DESCRIPTION

NXUserAborted() returns YES if the user pressed Command-period since the application last got an event in the main event loop, and NO if not. Command-period signals the user's intention to abort an ongoing process. Applications should call this function repeatedly during a modal session and respond appropriately if it ever returns YES.

NXResetUserAbort() resets the flag returned by **NXUserAborted()** to NO. It's called in the Application object's **run** method before getting each new event.

RETURN

NXUserAborted() returns YES if the user pressed Command-period, and NO otherwise.

NXUserName() → See NXHomeDirectory()

NXVPrintf() → See NXPutc()

NXVScanf() → See NXPutc()

NXWindowList() → See NXCountWindows()

NXWrite() → See NXRead()

NXWriteArray() → See NXReadArray()

NXWriteColor() → See NXReadColor()

NXWriteDefault() → See NXRegisterDefaults()

NXWriteDefaults() → See NXRegisterDefaults()

NXWriteObject() → See NXReadObject()

NXWriteObjectReference() → See NXReadObject()

NXWritePoint() → See NXReadPoint()

NXWriteRect() → See NXReadPoint()

NXWriteRootObject() → See NXReadObject()

NXWriteRootObjectToBuffer() → See NXReadObjectFromBuffer()

NXWriteSize() → See NXReadPoint()

NXWriteTIFF() → See NXReadTIFF()

NXWriteType() → See NXReadType()

NXWriteTypes() → See NXReadType()

NXWriteWordTable() → See NXReadWordTable()

NXYellowComponent() → See NXRedComponent()

NXZoneCalloc() → See NXZoneMalloc()

NXZoneFromPtr() → See NXZoneMalloc()

NXZoneFree() → See NXZoneMalloc()

NXZoneMalloc(), NXZoneCalloc(), NXZoneRealloc(), NXZoneFree(), NXDefaultMallocZone(), NXCreateZone(), NXCreateChildZone(), NXMergeZone(), NXDestroyZone(), NXZoneFromPtr(), NXZonePtrInfo(), NXMallocCheck(), NXNameZone()

SUMMARY Allocate memory

LIBRARY libsys_s.a

SYNOPSIS

#import <zone.h>

void ***NXZoneMalloc**(NXZone **zonep*, size_t *size*)
void ***NXZoneCalloc**(NXZone **zonep*, size_t *numElems*, size_t *byteSize*)
void ***NXZoneRealloc**(NXZone **zonep*, void **ptr*, size_t *size*)
void **NXZoneFree**(NXZone **zonep*, void **ptr*)
NXZone ***NXDefaultMallocZone**(void)
NXZone ***NXCreateZone**(size_t *startSize*, size_t *granularity*, int *canFree*)
NXZone ***NXCreateChildZone**(NXZone **parentZone*, size_t *startSize*,
    size_t *granularity*, int *canFree*)
void **NXMergeZone**(NXZone **zonep*)
void **NXDestroyZone**(NXZone **zonep*)
NXZone ***NXZoneFromPtr**(void **ptr*)
void **NXZonePtrInfo**(void **ptr*)
int **NXMallocCheck**(void)
void **NXNameZone**(NXZone **zonep*, const char **name*)

DESCRIPTION

These functions allocate and free memory space. They are similar to the standard C library **malloc**() functions, but allow the application writer more control over memory placement. By allocating frequently used objects from the same zone, the application writer can ensure better locality of reference; this can significantly improve performance on a paged virtual memory system. In other words, by grouping certain objects close together, you can ensure that consecutive references are less likely to result in memory paging activity.

To use these functions, you must first create a new zone using **NXCreateZone()**. You pass it a parameter *startSize*, which is the initial size of the new zone. The parameter *granularity* determines the granularity by which the zone itself grows and shrinks. If you are allocating a zone for small items, a good choice for both the initial size and granularity might be **vm_page_size**. The parameter *canFree* determines whether the allocator will free memory within the zone. If *canFree* is NO, memory cannot be freed and the allocator will be as fast as possible; but you will need to destroy the zone to reclaim the memory. You can call **NXCreateZone()** multiple times to create several zones. **NXCreateZone()** returns a pointer to the newly created zone.

NXZoneMalloc() allocates *size* bytes from the zone *zonep*, and returns a pointer to the allocated memory. **NXZoneCalloc()** allocates enough zeroed memory for *numElems* elements, each with a size of *byteSize* bytes from the zone *zonep*, and returns a pointer to the allocated memory. **NXZoneRealloc()** changes the size of the block pointed to by *ptr* to *size*. The block of memory may be moved, but its contents will be unchanged up to the lesser of the new and old sizes. All these functions return **NULL** upon failure.

NXCreateChildZone() creates a new zone which obtains memory from another zone. It returns a pointer to the new zone, or NX_NOZONE if you attempt to create a child zone from a zone which is itself a child. **NXMergeZone()** merges a child zone back into its parent zone. The allocated memory that was within the child zone remains valid.

NXZoneFree() returns memory to the zone from which it was allocated. **NXDestroyZone()** destroys a zone, and all the memory from the zone is reclaimed. **NXDefaultMallocZone()** returns the default zone. This is the zone used by the standard C library **malloc()** function. **NXZoneFromPtr()** returns the zone for a block of memory. The pointer *ptr* must have been returned from a prior malloc or realloc call. **NXZonePtrInfo()** will print information to **stdout** about the malloc block for the memory indicated by *ptr*. **NXMallocCheck()** verifies all internal malloc information, and returns zero if there is no error. **NXNameZone()** names the zone *zonep* with a copy of *name*.

NXZonePtrInfo() → See NXZoneMalloc()

NXZoneRealloc() → See NXZoneMalloc()

NX_ADDRESS()

SUMMARY Get a pointer to the objects stored in a List

LIBRARY libNeXT_s.a

SYNOPSIS

#import <objc/List.h>

id ***NX_ADDRESS**(List **aList*)

DESCRIPTION

This macro takes a List object *aList* as its argument and returns a pointer to the first **id** stored in the List. With this pointer, you get direct access to the contents of the List and can avoid the overhead of messaging. **NX_ADDRESS()** therefore provides an alternative to List's **objectAt:** method for situations where somewhat greater performance is required. In general, however, the method is the preferred way of accessing the List.

RETURN

This macro returns a pointer to the contents of a List object.

SEE ALSO

The specification for the List class.

NX_ASSERT()

SUMMARY Write an error message

LIBRARY libNeXT_s.a

SYNOPSIS

#import <appkit/nextstd.h>

void **NX_ASSERT**(int *exp*, char **msg*)

DESCRIPTION

This macro, which is defined in the header file **appkit/nextstd.h**, writes an error message if the program was compiled with the NX_BLOCKASSERTS flag undefined and if *exp* is false. The message *msg* is written to **stderr** if the application was launched from a terminal. If the application was launched by the Workspace Manager, the message is written using **syslog()** with the priority set to LOG_ERR. Normally,

syslog() writes messages to the Workspace Manager's console window. See the UNIX manual page for **syslog()** for more information about this function and how to write messages to places other than the console window.

If *exp* is true, no action is taken. Also, if the NX_BLOCKASSERTS flag is defined, a call to **NX_ASSERT()** has no effect.

NX_EVENTCODEMASK()

SUMMARY	Convert event type to mask
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <dpsclient/event.h>

int **NX_EVENTCODEMASK**(int *eventType*)

DESCRIPTION

This macro converts an event type, as defined in **dpsclient/event.h**, to an event mask. A window's event mask determines which types of events the Window Server will associate with the window.

An event mask is an **int** that stores a set of one-bit flags. (See **dpsclient/event.h** for a list of the predefined event masks.) By using **NX_EVENTCODEMASK()** to convert an event into an event mask, you can easily test an event's type. For example, assume *anEvent* is a pointer to an event record. You could find out if the record is for a keyboard event by converting its type to an event mask and comparing the mask to a mask for keyboard events:

```
if (NX_EVENTCODEMASK(anEvent->type) &
    (NX_KEYDOWNMASK|NX_KEYUPMASK|NX_FLAGSCHANGEDMASK)){
    /* anEvent is a keyboard event */
}
```

RETURN

This macro returns an integer mask.

NX_FREE() → See NX_MALLOC()

NX_HEIGHT() → See NX_X()

NX_MALLOC(), NX_REALLOC(), NX_FREE()

SUMMARY Allocate memory

LIBRARY libsys_s.a

SYNOPSIS

#import <appkit/nextstd.h>

type-name ***NX_MALLOC**(*type-name* **var*, *type-name*, int *num*)
type-name ***NX_REALLOC**(*type-name* **var*, *type-name*, int *num*)
void **NX_FREE**(void **pointer*)

DESCRIPTION

These macros allocate and free memory space by making calls to the standard C library functions **malloc()**, **realloc()**, and **free()**. For more information about these functions, see their UNIX manual pages.

NX_MALLOC() and **NX_REALLOC()** return a pointer of type *type* to the argument *var*. The amount of memory these two functions allocate is determined by multiplying *num* (which should be an **int**) by the number of bytes needed for the data type *type*. **NX_REALLOC()** should be used to change the size of the object *var*, just as **realloc** would be used. For convenience, these macros are shown below as they are defined in the header file **appkit/nextstd.h**:

```
#define  NX_MALLOC(VAR, TYPE, NUM)  \
    ((VAR) = (TYPE *) malloc((unsigned)(NUM)*sizeof(TYPE)))

#define  NX_REALLOC(VAR, TYPE, NUM)  \
    ((VAR) = (TYPE *) realloc((char *)(VAR), \
    (unsigned)(NUM)*sizeof(TYPE)))
```

NX_FREE() deallocates the space pointed to by *pointer*. It does nothing if *pointer* is NULL. It's also defined in **appkit/nextstd.h**, as shown below:

```
#define  NX_FREE(PTR)    free((char *) (PTR));
```

RETURN

NX_MALLOC() and **NX_REALLOC()** return pointers to the space they allocate or NULL if the request for space cannot be satisfied.

NX_MAXX() → See NX_X()

NX_MAXY() → See NX_X()

NX_MIDX() → See NX_X()

NX_MIDY() → See NX_X()

NX_PSDEBUG

SUMMARY	Print the current PostScript context
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <appkit/nextstd.h>

void **NX_PSDEBUG**

DESCRIPTION

NX_PSDEBUG prints the current Display PostScript context to the standard output device, along with the class, object, and method in which the macro appears. This macro works only when the application is compiled with DEBUG defined.

NX_RAISE(), NX_RERAISE(), NX_VALRETURN(), NX_VOIDRETURN

SUMMARY	Raise an exception
LIBRARY	libNeXT_s.a

SYNOPSIS

#import <objc/error.h >

void **NX_RAISE**(*int* code, const void **data1*, const void **data2*)
NX_RERAISE(void)
NX_VALRETURN(val)
NX_VOIDRETURN

DESCRIPTION

These macros initiate the error handling mechanism by alerting the appropriate error handler that an error has occurred. Error handlers exist in a nested hierarchy, which is created by using any number of nested NX_DURING...NX_ENDHANDLER constructs and by defining a top-level error handler.

The three arguments for **NX_RAISE()** provide information about the error condition. The first argument is a constant that acts as a label for the error. (Error codes used by the Application Kit are defined in the header file **appkit/errors.h**.) The next two arguments point to arbitrary data about the error. Within an NX_DURING...NX_ENDHANDLER construct, this data is stored in a local variable called **NXLocalHandler** (which is of type **NXHandler**, defined in the header file **streams/error.h**). (See the description of **NXAllocErrorData()** for more information about managing the storage of error data.) **NX_RAISE()** calls the function pointed to by **NXGetExceptionRaiser()**; see this function's description earlier in this chapter.

By default, an error handler should call **NX_RERAISE()** when it encounters an error that it can't handle, as shown below. **NX_RERAISE()** has the same functionality as **NX_RAISE()**, but it's called with no arguments. Since **NX_RERAISE()** implies a previous call to **NX_RAISE()**, the error data will already be stored in the local handler, eliminating the need for arguments.

```
NX_DURING
    /* code that may cause an error */
NX_HANDLER
    switch ( /* NXLocalHandler code */ )
    case
        NX_someErrorCode:
            /* code to execute for this type of error */
    default: NX_RERAISE();
NX_ENDHANDLER
```

NX_VALRETURN() and **NX_VOIDRETURN** can be used to exit a method or function from within the block of code between NX_DURING and NX_HANDLER labels. The only legal ways of exiting this block are falling out the bottom or using one of these macros. **NX_VALRETURN()** causes its method (or function) to return *val*, while **NX_VOIDRETURN** can be used to return from a method (or function) that has no return value. Use these macros only within an NX_DURING...NX_HANDLER construct.

SEE ALSO

NXAllocErrorData(), **NXSetUncaughtExceptionHandler()**, **NXDefaultTopLevelErrorHandler()**, **NXRegisterErrorReporter()**, **NXDefaultExceptionRaiser()**

NX_REALLOC() → See NX_MALLOC()

NX_RERAISE() → See NX_RAISE()

NX_VALRETURN() → See NX_RAISE()

NX_VOIDRETURN() → See NX_RAISE()

NX_WIDTH() → See NX_X()

NX_X(), NX_Y(), NX_WIDTH(), NX_HEIGHT(), NX_MAXX(), NX_MAXY(), NX_MIDX(), NX_MIDY()

SUMMARY Query an NXRect structure

LIBRARY libNeXT_s.a

SYNOPSIS

#import <appkit/graphics.h>

NXCoord **NX_X**(NXRect **aRect*)
NXCoord **NX_Y**(NXRect **aRect*)
NXCoord **NX_WIDTH**(NXRect **aRect*)
NXCoord **NX_HEIGHT**(NXRect **aRect*)
NXCoord **NX_MAXX**(NXRect **aRect*)
NXCoord **NX_MAXY**(NXRect **aRect*)
NXCoord **NX_MIDX**(NXRect **aRect*)
NXCoord **NX_MIDY**(NXRect **aRect*)

DESCRIPTION

These macros return information about the NXRect structure referred to by *aRect*. An NXRect structure is defined by a point that locates the rectangle (x and y coordinates) and an extent that determines its size (a width and height as measured along the x- and y-axes).

RETURN

NX_X() and **NX_Y**() return the x and y coordinates that locate the rectangle. These will be the smallest coordinate values within the rectangle.

NX_HEIGHT() and **NX_WIDTH**() return the width and height of the rectangle.

NX_MAXX() and **NX_MAXY**() return the largest x and y coordinates in the rectangle. These are calculated by adding the width of the rectangle to the x coordinate returned by **NX_X**() and by adding the height of the rectangle to the y coordinate returned by **NX_Y**().

NX_MIDX() and **NX_MIDY**() return the x and y coordinates that lie at the center of the rectangle, exactly midway between the smallest and largest coordinate values.

SEE ALSO

NXSetRect()

NX_Y() → See NX_X()

NX_ZONEMALLOC(), NX_ZONEREALLOC()

SUMMARY Allocate zone memory

LIBRARY libsys_s.a

SYNOPSIS

#import <appkit/nextstd.h>

type-name ***NX_ZONEMALLOC**(NXZone *zone*, *type-name* **var*, *type-name*, int *num*)
type-name ***NX_ZONEREALLOC**(NXZone *zone*, *type-name* **var*, *type-name*, int *num*)

DESCRIPTION

These macros allocate and free memory space by making calls to the functions **NXZoneMalloc()** and **NXZoneRealloc()**. For more information about these functions, see their descriptions earlier in this chapter.

NX_ZONEMALLOC() and **NX_ZONEREALLOC()** return a pointer of type *type-name* to the argument *var* allocated in *zone*. The amount of memory these two macros allocate is determined by multiplying *num* (which should be an **int**) by the number of bytes needed for the data type *type-name*. **NX_ZONEREALLOC()** should be used to change the size of the object *var*, just as **realloc()** or **NXZoneRealloc()** would be used. For convenience, these macros are shown below as they are defined in the header file **appkit/nextstd.h**:

```
#define  NX_ZONEMALLOC(Z, VAR, TYPE, NUM)  \
    ((VAR) = (TYPE *) NXZoneMalloc((Z), \
    (unsigned)(NUM)*sizeof(TYPE)))

#define  NX_ZONEREALLOC(Z, VAR, TYPE, NUM)  \
    ((VAR) = (TYPE *) NXZoneRealloc((Z), (char *)(VAR), \
    (unsigned)(NUM)*sizeof(TYPE)))
```

RETURN

NX_ZONEMALLOC() and **NX_ZONEREALLOC()** return pointers to the space they allocate or NULL if the request for space cannot be satisfied.

Single-Operator Functions

The Display PostScript system provides a C function interface for each operator in the PostScript language. These functions let you easily execute individual PostScript operators from your application. Adobe Systems Incorporated provides the primary documentation for these operators and for **pswrap**, the utility that creates a C function for one or more PostScript operators. (See "Suggested Reading" in the *Technical Summaries* manual for **pswrap** and other Display PostScript system documentation.)

NeXT has added several operators and their corresponding single-operator functions to the basic Display PostScript system. The operators are documented in Chapter 4, "PostScript Operators," and the functions are listed below. These functions are provided in the library **libNeXT_s.a**.

In the Display PostScript system, each PostScript operator is represented by two single-operator functions (or "procedures," as they are referred to in Adobe documentation), one that takes a context argument and another that assumes the current PostScript context. The functions that take a context argument have a "DPS" prefix; those that assume the current context have a "PS" prefix. For example, the **moveto** operator is represented by these functions:

> **DPSmoveto**(DPSContext *context*, float *x*, float *y*)
> **PSmoveto**(float *x*, float *y*)

To save space, only the single-operator functions prefixed with "PS" are listed here. The header file **dpsclient/dpswraps.h** declares the function prototypes for all single-operator functions having the "DPS" prefix; the header file **dpsclient/wraps.h** declares the prototypes for "PS" functions.

Operand names available in the PostScript language, such as Copy or Sover for the **composite** operator, are defined as symbolic constants for use from C, but in all uppercase and preceded by "NX_" (for example, NX_COPY and NX_SOVER). These symbolic constants are defined in the NeXT header file **dpsNeXT.h**, except for the event-related ones, which are in **dpsclient/event.h** and **appkit/appkit.h**.

As with the basic Display PostScript single-operator functions, some of the C functions listed below have parameters that match the operands of their corresponding PostScript operators. For example, the **setalpha** operator accepts a number on the PostScript operand stack, while the C function **PSsetalpha**() takes a float as an argument. The functions may also have parameters that point to returned values, corresponding to results returned on the operand stack by the PostScript operator. The **buttondown** operator returns a Boolean on the stack indicating whether the left mouse button is down; **PSbuttondown**() has a parameter that's a pointer to a Boolean, which upon return will contain 1 or 0 to indicate the status of the mouse button.

Other C functions have no parameters where their corresponding PostScript operators expect operands or leave results on the operand stack. These functions assume that they'll be called with the appropriate objects already on the operand stack, and they'll leave any PostScript objects they generate on the operand stack instead of returning them through parameters. For example, the **PSalphaimage()** function requires that you place the appropriate operands on the operand stack before calling the function. You can learn which operands the function expects by looking at the declaration of the corresponding operator.

To support the functions that use the operand stack rather than parameters, the Display PostScript system has several additional functions for putting values on and getting values off the stack:

Function	**Effect**

PSsendint()
PSsendfloat()
PSsendboolean()
PSsendstring()
	Puts a single value of the specified type on the operand stack

PSgetint()
PSgetfloat()
PSgetboolean()
PSgetstring()
	Gets a single value of the specified type from the operand stack

PSsendintarray()
PSsendfloatarray()
PSsendchararray()
	Puts a series of objects on the operand stack

PSgetintarray()
PSgetfloatarray()
PSgetchararray()
	Gets a series of objects from the operand stack

Note the following:

- In addition to the standard C types, **pswrap** uses two others: **boolean** and **userobject**. A **boolean** variable is an **int** having either a zero or a nonzero value. The zero value is equivalent to the PostScript value *false*, and the nonzero value is equivalent to the PostScript value *true*. The **userobject** type is an **int** that refers to the value returned by **DPSDefineUserObject()**. See *Extensions for the Display PostScript System* for more information on user objects.

- Functions that require a graphics state userobject parameter can use the constant NXNullObject to refer to the current graphics state. NXNullObject is declared in **appkit/Application.h**.

- Functions that pass an array as a parameter include an additional parameter indicating the size of the array. The size parameter is used only by **pswrap** and is not sent to the Window Server. It's your responsibility to provide enough space for the array's data.

If a function listed here is set up inconveniently for your purposes, you can always use **pswrap** to make your own.

Warning: Those functions marked "/* Internal */" below are reserved for use by the Application Kit. Only call them in applications that don't make use of the Kit.

void **PSadjustcursor(**float *dx*, float *dy*)

void **PSalphaimage**(void)

void **PSbasetocurrent**(float *x*, float *y*, float **px*, float **py*)

void **PSbasetoscreen(**float *x*, float *y*, float **px*, float **py*)

void **PSbuttondown**(boolean **pflag*)

void **PScleartrackingrect(**int *trectNum*, userobject *gstate*)

void **PScomposite**(float *x*, float *y*, float *width*, float *height*, userobject *srcGstate*, float $dest_x$, float $dest_y$, int *op*)

op values:

NX_CLEAR
NX_COPY
NX_SOVER
NX_DOVER
NX_SIN
NX_DIN
NX_SOUT
NX_DOUT
NX_SATOP
NX_DATOP
NX_XOR
NX_PLUSD
NX_PLUSL

void **PScompositerect**(float $dest_x$, float $dest_y$, float *width*, float *height*, int *op*)

op values: **PScompositerect**() supports NX_HIGHLIGHT in addition to the values listed under **PScomposite**().

void **PScountframebuffers**(int **pcount*)

void **PScountscreenlist**(int *context*, int **pcount*)

void **PScountwindowlist**(int *context*, int **pcount*)

void **PScurrentactiveapp**(int **pcontext*)   /* Internal */

void **PScurrentalpha**(float **pcoverage*)

void **PScurrentdefaultdepthlimit**(int **plimit*)

void **PScurrentdeviceinfo**(userobject *window*, int **pminbps*, int **pmaxbps*, int **pcolor*) void **PScurrenteventmask**(userobject *window*, int **pmask*)   /* Internal */

void **PScurrentmouse**(userobject *window*, float **px*, float **py*)   /* Internal */

void **PScurrentowner**(userobject *window*, int **pcontext*)

void **PScurrentrusage**(float **pnow*, float **puTime*, float **psTime*, int **pmsgSend*, int **pmsgRcv*, int **pnSignals*, int **pnVCSw*, int **pnIvCSw*)

void **PScurrenttobase**(float *x*, float *y*, float **px*, float **py*)

void **PScurrenttoscreen**(float *x*, float *y*, float **px*, float **py*)

void **PScurrentuser**(int **puid*, int **pgid*)

void **PScurrentwaitcursorenabled**(boolean **pflag*)

void **PScurrentwindow**(int **pnum*)

void **PScurrentwindowalpha**(userobject *window*, int **palpha*)

void **PScurrentwindowbounds**(userobject *window*, float **px*, float **py*, float **pwidth*, float **pheight*)

void **PScurrentwindowdepth**(userobject *window*, int **pdepth*)

void **PScurrentwindowdepth**(userobject *window*, int **plimit*)

void **PScurrentwindowdict**(userobject *window*)   /* Internal */

void **PScurrentwindowlevel**(userobject *window*, int **plevel*)

void **PScurrentwriteblock**(int **pflag*)

void **PSdissolve**(float src_x, float src_y, float *width*, float *height*, userobject *srcGstate*, float $dest_x$, float $dest_y$, float *delta*)

void **PSdumpuserobjects**(void)

void **PSdumpwindow**(int *level*, userobject *window*)   /* Internal */

void **PSdumpwindows**(int *level*, userobject *context*)   /* Internal */

void **PSfindwindow**(float *x*, float *y*, int *place*, userobject *otherWin*, float **px*, float **py*, int **pwinFound*, boolean **pdidFind*)

place values:

NX_ABOVE
NX_BELOW

void **PSflushgraphics**(void)

void **PSframebuffer**(int *index*, int *nameLen*, char *name[]*, int **pslot*, int **punit*,
    int **pROMid*, int **px*, int **py*, int **pw*, int **ph*, int **pdepth*)

void **PSfrontwindow**(int **pnum*)   /* Internal */

void **PShidecursor**(void)

void **PShideinstance**(float *x*, float *y*, float *width*, float *height*)

void **PSmachportdevice**(int *w*, int *h*, int *bbox*[], int *bboxSize*, float *matrix*[], char **phost*,
    char **pport*, char **ppixelDict*)

void **PSmovewindow**(float *x*, float *y*, userobject *window*)   /* Internal */

void **PSnewinstance**(void)

void **PSnextrelease**(int *size*, char *string*[])
    /* *size* is the maximum number of characters copied into *string* */

void **PSobscurecursor**(void)

void **PSorderwindow**(int *place*, userobject *otherWindow*, int *window*)   /* Internal */

place values:

NX_ABOVE
NX_BELOW
NX_OUT

void **PSosname**(int *size*, char *string*[])
    /* *size* is the maximum number of characters copied into *string* */

void **PSostype**(int **ptype*)

void **PSplacewindow**(float *x*, float *y*, float *width*, float *height*, userobject *window*)
    /* Internal */

void **PSplaysound**(char **name*, int *priority*)

void **PSposteventbycontext**(int *type*, float *x*, float *y*, int *time*, int *flags*, int *window*, int
    subtype, int *data1*, int *data2*, int *context*, boolean **psuccess*)

void **PSreadimage**(void)

void **PSrevealcursor**(void)

void **PSrightbuttondown**(int **pflag*) void **PSrightstilldown**(int *eventnum*, boolean **pflag*)

void **PSscreenlist**(int *context*, int *count*, int *windows*[])

void **PSscreentobase**(float *x*, float *y*, float **px*, float **py*)

void **PSscreentocurrent**(float *x*, float *y*, float **px*, float **py*)

void **PSsetactiveapp**(int *context*) /* Internal */

void **PSsetalpha**(float *coverage*)

void **PSsetautofill**(boolean *flag*, userobject *window*)

void **PSsetcursor**(float *x*, float *y*, float *mx*, float *my*)

void **PSsetdefaultdepthlimit**(int *limit*)

void **PSseteventmask**(int *mask*, userobject *window*) /* Internal */

mask values:

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

void **PSsetexposurecolor**(void)

void **PSsetflushexposures**(boolean *flag*)

void **PSsetinstance**(boolean *flag*)

void **PSsetmouse**(float *x*, float *y*)

void **PSsetowner**(userobject *owner*, userobject *window*)

void **PSsetpattern**(userobject *patternDict*) void **PSsetsendexposed**(boolean *flag*, userobject *window*) /* Internal */

void **PSsettrackingrect**(float *x*, float *y*, float *width*, float *height*, boolean *leftFlag*, boolean *rightFlag*, boolean *inside*, int *userData,* int *trectNum*, userobject *gstate*)

void **PSsetwaitcursorenabled**(boolean *flag*)

void **PSsetwindowdepthlimit**(int *limit*, userobject *window*)

void **PSsetwindowdict**(userobject *window*) /* Internal */

void **PSsetwindowlevel**(int *level*, userobject *window*)

void **PSsetwindowtype**(int *type*, userobject *window*)

void **PSsetwriteblock**(int *flag*)

void **PSshowcursor**(void)

void **PSsizeimage**(float *x*, float *y*, float *width*, float *height*, int **pwidth*, int **pheight*, int **pbitsPerComponent*, float *matrix*[], boolean **pmultiproc*, int **pnColors*)

void **PSstilldown**(int *eventnum*, boolean **pflag*)

void **PStermwindow**(userobject *window*) /* Internal */

void **PSwindow**(float *x*, float *y*, float *width*, float *height*, int *type*, int **pwindow*) /* Internal */

void **PSwindowdevice**(userobject *window*)

void **PSwindowdeviceround**(userobject *window*)

void **PSwindowlist**(int *context*, int *count*, int *windows[]*)

Run-Time Functions

This section describes functions and macros that are part of NeXT's run-time system for the Objective-C language. Some, such as **sel_getUid()** and **objc_loadModules()**, might be useful when called within an Objective-C program, but most are provided mainly to make it possible to define other interfaces to the run-time system. For most programs, Objective-C is itself a sufficient and complete interface to the run-time system; the messages and class definitions in Objective-C source files are compiled to execute correctly at run time without the aid of additional function calls.

The functions described here are divided into five groups, each with its own prefix:

- The basic run-time functions have an "objc_" prefix.
- Functions that operate on class objects have a "class_" prefix and take as their first argument a structure of type **Class**. **Class** is the defined type (in **objc/objc.h**) for class objects. However, to receive messages in Objective-C source code, class objects must be of type **id**, so **id** rather than **Class** is the type generally used in Objective-C programs.
- Functions that operate on instances have an "object_" prefix and take as their first argument the **id** of the instance.
- Functions that give information about method selectors have a "sel_" prefix.
- Functions that describe method implementations have a "method_" prefix.

NeXT reserves these prefixes for functions in the run-time system.

In addition to these functions, there are also a few macros that operate on the values passed in a message. They begin with a "marg_" prefix (for "message argument").

class_addClassMethods() → See class_getInstanceMethod()

class_addInstanceMethods() → See class_getInstanceMethod()

class_createInstance(), class_createInstanceFromZone()

SUMMARY Create a new instance of a class

LIBRARY libsys_s.a

SYNOPSIS

#import <objc/objc-class.h>

id **class_createInstance**(Class *aClass*, unsigned int *indexedIvarBytes*)
id **class_createInstanceFromZone**(Class *aClass*, unsigned int *indexedIvarBytes*,
  NXZone **zone*)

DESCRIPTION

These functions provide an interface to the object allocators used by the run-time system. The default allocators, which can be changed by reassigning the **_alloc** and **_zoneAlloc** variables, create a new instance of *aClass*, initialize its **isa** instance variable to point to the class, and return the new instance. All other instance variables are initialized to 0.

The two functions are identical, except that **class_createInstanceFromZone()** allocates memory for the new object from the region specified by *zone*; **class_createInstance()** doesn't specify a zone. Object's **new** method uses **class_createInstance()** to allocate memory for a new object. The **alloc** and **allocFromZone:** methods use **class_createInstanceFromZone()**, with **alloc** taking the memory from the default zone returned by **NXDeaultMallocZone()**.

The second argument to both functions, *indexedIvarBytes*, states the number of extra bytes required for indexed instance variables. Normally, it's 0.

Indexed instance variables are instance variables that don't have a fixed size; usually they're arrays whose length can't be computed at compile time. Since the components of a C structure can't be of uncertain size, indexed instance variables can't be declared in the class interface. The class must account for them outside the normal channels provided by the Objective-C language.

All of the storage required for indexed instance variables must be allocated through this function. The following code shows how it might be used in an instance-creating class method:

```
+ new:(unsigned int)numBytes
{
    self = class_createInstance((Class)self, numBytes);
    length = numBytes;
    . . .
}

- (char *)getArray
{
    return(object_getIndexedIvars(self));
}
```

Indexed instance variables should be avoided if at all possible. It's a much better practice to store variable-length data outside the object and declare one real instance variable that points to it and perhaps another that records its length. For example:

```
+ new:(unsigned int)numBytes
{
    self = [super new];
    data = malloc(numBytes);
    length = numBytes;
    . . .
}

- (char *)getArray
{
    return data;
}
```

RETURN

Both functions return a new instance of *aClass*.

class_createInstanceFromZone() → See class_createInstance()

class_getClassMethod() → See class_getInstanceMethod()

class_getInstanceMethod(), class_getClassMethod(), class_addInstanceMethods(), class_addClassMethods(), class_removeMethods()

SUMMARY Get, add, and remove methods for the class

LIBRARY libsys_s.a

SYNOPSIS

#import <objc/objc-class.h>

Method **class_getInstanceMethod**(Class *aClass*, SEL *aSelector*)
Method **class_getClassMethod**(Class *aClass*, SEL *aSelector*)
void **class_addInstanceMethods**(Class *aClass*, struct objc_method_list **methodList*)
void **class_addClassMethods**(Class *aClass*, struct objc_method_list **methodList*)
void **class_removeMethods**(Class *aClass*, struct objc_method_list **methodList*)

DESCRIPTION

The first two functions, **class_getInstanceMethod**() and **class_getClassMethod**(), return a pointer to the class data structure that describes the *aSelector* method. For **class_getInstanceMethod**(), *aSelector* must identify an instance method; for **class_getClassMethod**(), it must identify a class method. Both functions return a NULL pointer if *aSelector* doesn't identify a method defined in or inherited by *aClass*.

The run-time system uses the next two functions, **class_addInstanceMethods**() and **class_addClassMethods**(), to implement Objective-C categories. Each function adds the methods in *methodList* to the dictionary of methods defined for *aClass*. **class_addInstanceMethods**() adds methods that can be used by instances of the class and **class_addClassMethods**() adds methods used by the class object. Before adding a method, both functions map the method name to a SEL selector and check for duplicates. A warning is sent to the standard error stream if any ambiguities exist.

The last function, **class_removeMethods**(), removes the methods in *methodList* from *aClass*. It can remove both class and instance methods.

RETURN

class_getInstanceMethod() and **class_getClassMethod**() return a pointer to the data structure that describes the *aSelector* method as implemented for *aClass*.

class_getInstanceVariable()

SUMMARY	Get the class template for an instance variable
LIBRARY	libsys_s.a

SYNOPSIS

#import <objc/objc-class.h>

Ivar **class_getInstanceVariable**(Class *aClass*, STR *variableName*)

RETURN

This function returns a pointer to the class data structure that describes the *variableName* instance variable. If *aClass* doesn't define or inherit the instance variable, a NULL pointer is returned.

class_getVersion() → See class_setVersion()

class_poseAs()

SUMMARY	Pose as the superclass
LIBRARY	libsys_s.a

SYNOPSIS

#import <objc/objc-class.h>

Class **class_poseAs**(Class *theImposter*, Class *theSuperclass*)

DESCRIPTION

class_poseAs() causes one class, *theImposter*, to take the place of its own superclass, *theSuperclass*. Messages sent to *theSuperclass* will actually be received by *theImposter*. The posing class can't declare any new instance variables, but it can define new methods and even override methods defined in the superclass.

Posing is usually done through Object's **poseAs:** method, which calls this function.

RETURN

class_poseAs() returns its first argument, *theImposter*.

class_removeMethods() → See class_getInstanceMethod()

class_setVersion(), class_getVersion()

SUMMARY Set and get the class version

LIBRARY libsys_s.a

SYNOPSIS

#import <objc/objc-class.h>

void **class_setVersion**(Class *aClass*, int *theVersion*)
int **class_getVersion**(Class *aClass*)

DESCRIPTION

These functions set and return the class version number. This number is used when archiving instances of the class.

Object's **setVersion:** and **version** methods do the same work as these functions.

RETURN

class_getVersion() returns the version number for *aClass* last set by **class_setVersion**().

marg_getRef() → See marg_getValue()

marg_getValue(), marg_getRef(), marg_setValue()

SUMMARY Examine and alter method argument values

LIBRARY libsys_s.a

SYNOPSIS

#import <objc/objc-class.h>

type-name **marg_getValue**(marg_list *argFrame*, int *offset*, *type-name*)
type-name ***marg_getRef**(marg_list *argFrame*, int *offset*, *type-name*)
void **marg_setValue**(marg_list *argFrame*, int *offset*, *type-name*, *type-name value*)

DESCRIPTION

These three macros get and set the values of arguments passed in a message. They're designed to be used within implementations of the **forward::** method, which is described under the Object class in Chapter 2, "Class Specifications."

The first argument to each macro, *argFrame*, is a pointer to the list of arguments passed in the message. The run-time system passes this pointer to the **forward::** method, making it available to be used in these macros. The next two arguments—an *offset* into the argument list and the type of the argument at that offset—can be obtained by calling **method_getArgumentInfo()**

The first macro, **marg_getValue**, returns the argument at *offset* in *argFrame*. The return value, like the argument, is of type *type-name*. The second macro, **marg_getRef**, returns a reference to the argument at *offset* in *argFrame*. The pointer returned is to an argument of the *type-name* type. The third macro, **marg_setValue**, alters the argument at *offset* in *argFrame* by assigning it *value*. The new value must be of the same type as the argument.

Since **method_getArgumentInfo()** encodes the argument type according to the conventions of the **@encode()** compiler directive, the type must first be expanded to a full type name before it can be used in these macros. The offset provided by **method_getArgumentInfo()** can be passed directly to the macros without change.

RETURN

marg_getValue returns a *type-name* argument value. **marg_getRef** returns a pointer to a *type-name* argument value.

marg_setValue() → See marg_getValue()

method_getArgumentInfo() → See method_getNumberOfArguments()

method_getNumberOfArguments(), method_getSizeOfArguments(), method_getArgumentInfo()

SUMMARY	Get information about a method
LIBRARY	libsys_s.a

SYNOPSIS

#import <objc/objc-class.h>

unsigned int **method_getNumberOfArguments**(Method *aMethod*)
unsigned int **method_getSizeOfArguments**(Method *aMethod*)
unsigned int **method_getArgumentInfo**(Method *aMethod*, int *index*, char ***type*, int **offset*)

DESCRIPTION

The three functions described here all provide information about the argument structure of a particular method. They take as their first argument the method's data structure, *aMethod*, which can be obtained by calling **class_getInstanceMethod**() or **class_getClassMethod**().

The first function, **method_getNumberOfArguments**(), returns the number of arguments that *aMethod* takes. This will be at least two, since it includes the "hidden" arguments, **self** and **_cmd**, which are the first two arguments passed to every method implementation.

The second function, **method_getSizeOfArguments**(), returns the number of bytes that all of *aMethod*'s arguments, taken together, would occupy on the stack. This information is required by **objc_msgSendv**().

The third function, **method_getArgumentInfo**(), takes an *index* into *aMethod*'s argument list and returns, by reference, the type of the argument and the offset to the location of that argument in the list. Indices begin with 0. The "hidden" arguments **self** and **_cmd** are indexed at 0 and 1; method-specific arguments begin at index 2. The offset is measured in bytes and depends on the size of arguments preceding the indexed argument in the argument list. The type is encoded according to the conventions of the **@encode**() compiler directive.

The information obtained from **method_getArgumentInfo**() can be used in the **marg_getValue**, **marg_getRef**, and **marg_setValue** macros to examine and alter the values of an argument on the stack after *aMethod* has been called. The offset can be passed directly to these macros, but the type must first be decoded to a full type name.

RETURN

method_getNumberOfArguments() returns how many arguments the implementation of *aMethod* takes, and **method_getSizeOfArguments**() returns how many bytes the arguments take up on the stack. **method_getArgumentInfo**() returns the *index* it is passed.

method_getSizeOfArguments() → See method_getNumberOfArguments()

objc_addClass() → See objc_getClass()

objc_getClass(), objc_getMetaClass(), objc_getClasses(), objc_addClass(), objc_getModules()

SUMMARY	Manage run-time structures
LIBRARY	libsys_s.a

SYNOPSIS

#import <objc/objc-runtime.h>

id **objc_getClass**(STR *aClassName*)
id **objc_getMetaClass**(STR *aClassName*)
NXHashTable ***objc_getClasses**(void)
void **objc_addClass**(Class *aClass*)
Module ***objc_getModules**(void)

DESCRIPTION

These functions return and modify the principal data structures used by the run-time system.

objc_getClass() returns the **id** of the class object for the *aClassName* class, and **objc_getMetaClass**() returns the **id** of the metaclass object for the *aClassName* class. The metaclass object holds information used by the class object, just as the class object holds information used by instances of the class. Both functions print a message to the standard error stream if *aClassName* isn't part of the executable image.

objc_getClasses() returns a pointer to a hash table containing all the Objective-C classes that are currently part of the executable image. The NXHashTable return type is defined in the **objc/hashtable.h** header file. **objc_addClass**() adds *aClass* to the list of currently loaded classes.

The compiler creates a Module data structure for each file it compiles. The **objc_getModules**() function returns a pointer to a list of all the modules that are part of the executable image.

RETURN

objc_getClass() and **objc_getMetaClass**() return the class and metaclass objects for *aClassName*. **objc_getClasses**() returns a pointer to a hash table of all current classes, and **objc_getModules**() returns a pointer to all current modules.

objc_getClasses() → See objc_getClass()

objc_getMetaClass() → See objc_getClass()

objc_getModules() → See objc_getClass()

objc_loadModules(), objc_unloadModules()

SUMMARY	Dynamically load and unload classes
LIBRARY	libsys_s.a

SYNOPSIS

#import <objc/objc-load.h>

long **objc_loadModules**(char **files*[], NXStream **stream*,
void (**callback*)(Class, Category), struct mach_header ***header*,
char **debugFilename*)
long **objc_unloadModules**(NXStream **stream*, void (**callback*)(Class, Category))

DESCRIPTION

objc_loadModules() dynamically loads object files containing Objective-C class and category definitions into a running program. Its first argument, *files*, is a list of null-terminated pathnames for the object files containing the classes and categories that are to be loaded. They can be full paths or paths relative to the current working directory. The second argument, *stream*, is a pointer to an NXStream where any error messages produced by the loader will be written. It can be NULL, in which case no messages will be written.

The third argument, *callback*, allows you to specify a function that will be called immediately after each class or category is loaded. When a category is loaded, the function is passed both the **Category** structure and the **Class** structure for that category. When a class is loaded, it's passed only the **Class** structure. Like *stream*, *callback* can be NULL.

The fourth argument, *header*, is used to get a pointer to the **mach_header** structure for the loaded modules. It, too, can be NULL. All the modules in *files* are grouped under the same header.

The final argument, which also can be NULL, is the pathname for a file that the loader will create and initialize with a copy of the loaded modules. This file can be passed to the debugger and added to the executable image that it's debugging. For example:

```
(gdb) add-file debugFilename
```

obj_unloadModules() unloads all the modules loaded by **objc_loadModules**(), that is, all the modules from the *files* list. Each time it's called, it unloads another set of modules, working its way back from the modules loaded by the most recent call to **objc_loadModules**() to those loaded by the next most recent call, and so on.

The first argument to **obj_unloadModules**(), *stream*, is a pointer to an NXStream where error messages will be written. Its second argument, *callback*, allows you to specify a function that will be called immediately before each class or category is unloaded. Both arguments can be NULL.

RETURN

Both functions return 0 if the modules are successfully loaded or unloaded and 1 if they're not.

objc_msgSend(), objc_msgSendSuper(), objc_msgSendv()

SUMMARY	Dispatch messages at run time
LIBRARY	libsys_s.a

SYNOPSIS

#import <objc/objc-runtime.h>

id **objc_msgSend**(id *theReceiver*, SEL *theSelector*, ...)
id **objc_msgSendSuper**(struct objc_super **superContext*, SEL *theSelector*, ...)
id **objc_msgSendv**(id *theReceiver*, SEL *theSelector*, unsigned int *argSize*,
    marg_list *argFrame*)

DESCRIPTION

The compiler converts every message expression into a call on one of the first two of these three functions. Messages to **super** are converted to calls on **objc_msgSendSuper**(); all others are converted to calls on **objc_msgSend**().

Both functions find the implementation of the *theSelector* method that's appropriate for the receiver of the message. For **objc_msgSend**(), *theReceiver* is passed explicitly as an argument. For **objc_msgSendSuper**(), *superContext* defines the context in which the message was sent, including who the receiver is.

Arguments that are included in the *aSelector* message are passed directly as additional arguments to both functions.

Calls to **objc_msgSend**() and **objc_msgSendSuper**() should be generated only by the compiler. You shouldn't call them directly in the Objective-C code you write.

The third function, **objc_msgSendv()**, is an alternative to **objc_msgSend()** that's designed to be used within class-specific implementations of the **forward::** method. Instead of being passed each of the arguments to the *aSelector* message, it takes a pointer to the arguments list, *argFrame*, and the size of the list in bytes, *argSize*. *argSize* can be obtained by calling **method_getArgumentSize()**; *argFrame* is passed as the second argument to the **forward::** method.

objc_msgSendv() parses the argument list based on information stored for *aSelector* and the class of the receiver. Because of this additional work, it's more expensive than **objc_msgSend()**.

RETURN

Each method passes on the value returned by the *aSelector* method.

objc_msgSendSuper() → See objc_msgSend()

objc_msgSendv() → See objc_msgSend()

objc_unloadModules() → See objc_loadModules()

object_copy() → See object_dispose()

object_copyFromZone() → See object_dispose()

object_dispose(), object_copy(), object_realloc(), object_copyFromZone(), object_reallocFromZone()

SUMMARY	Manage object memory
LIBRARY	libsys_s.a

SYNOPSIS

#import <objc/Object.h>

id **object_dispose**(Object **anObject*)
id **object_copy**(Object **anObject*, unsigned int *indexedIvarBytes*)
id **object_realloc**(Object **anObject*, unsigned int *numBytes*)
id **object_copyFromZone**(Object **anObject*, unsigned int *indexedIvarBytes*,
    NXZone **zone*)
id **object_reallocFromZone**(Object **anObject*, unsigned int *numBytes*,
    NXZone **zone*)

DESCRIPTION

These five functions, along with **class_createInstance()** and **class_createInstanceFromZone()**, manage the dynamic allocation of memory for objects. Like those two functions, each of them is simply a "cover" for—a way of calling—another, private function.

object_dispose() frees the memory occupied by *anObject* after setting its **isa** instance variable to **nil**, and returns **nil**. The function it calls to do this work can be changed by reassigning the **_dealloc** variable.

object_copy() and **object_copyFromZone()** create a new object that's an exact copy of *anObject* and return the new object. The second argument to both functions, *indexedIvarBytes*, specifies the number of additional bytes that should be allocated for the copy to accommodate indexed instance variables; it serves the same purpose as the second argument to **class_createInstance()**. The functions that **object_copy()** and **object_copyFromZone()** call to do this work can be changed by reassigning the **_copy** and **_zoneCopy** variables.

object_realloc() and **object_reallocFromZone()** reallocate storage for *anObject*, adding *numBytes* if possible. The memory previously occupied by *anObject* is freed if it can't be reused, and a pointer to the new location of *anObject* is returned. The functions that **object_realloc()** and **object_reallocFromZone()** call to do this work can be changed by reassigning the **_realloc** and **_zoneRealloc** variables.

The Object class defines a method interface for the first three of these functions. The **free** instance method corresponds to **object_dispose()**. And the **copy** and **copyFromZone:** methods correspond to **object_copy()** and **object_copyFromZone()**.

RETURN

object_dispose() returns **nil**, **object_copy()** and **object_copyFromZone()** return the copy, and **object_realloc()** and **object_reallocFromZone()** return the reallocated object.

object_getClassName()

SUMMARY Return the class name

LIBRARY libsys_s.a

SYNOPSIS

#import <objc/objc.h>

STR **object_getClassName**(id *anObject*)

DESCRIPTION

This function returns the name of *anObject*'s class. *anObject* should be an instance object, not a class object.

object_getIndexedIvars()

SUMMARY Return a pointer to an object's extra memory

LIBRARY libsys_s.a

SYNOPSIS

#import <objc/objc.h>

void ***object_getIndexedIvars**(id *anObject*)

RETURN

object_getIndexedIvars() returns a pointer to the first indexed instance variable of *anObject*, or NULL if *anObject* has no indexed instance variables.

SEE ALSO

class_createInstance()

object_getInstanceVariable() → See object_setInstanceVariable()

object_realloc() → See object_dispose()

object_reallocFromZone() → See object_dispose()

object_setInstanceVariable(), object_getInstanceVariable()

SUMMARY	Set and get instance variables
LIBRARY	libsys_s.a

SYNOPSIS

#import <objc/Object.h>

Ivar **object_setInstanceVariable**(id *anObject*, STR *variableName*, void **value*)
Ivar **object_getInstanceVariable**(id *anObject*, STR *variableName*, void ***valuePtr*)

DESCRIPTION

object_setInstanceVariable() assigns a new value to the *variableName* instance variable belonging to *anObject*. The new value is passed in the third argument, *value*. **object_getInstanceVariable()** gets the value of *anObject*'s *variableName* instance variable. The value is returned by reference through the third argument, *valuePtr*.

These functions provide a way of setting and getting instance variables, without having to implement methods for that purpose. For example, Interface Builder calls **object_setInstanceVariable()** to initialize programmer-defined "outlet" instance variables.

RETURN

Both functions return a pointer to the class template that describes the *variableName* instance variable. A NULL pointer is returned if *anObject* has no instance variable with that name.

The returned template has a field describing the data type of the instance variable. You can check it to be sure that the value set is of the correct type.

sel_getName() → See sel_getUid()

sel_getUid(), sel_getName()

SUMMARY Match method selectors with method names

LIBRARY libsys_s.a

SYNOPSIS

#import <objc/objc.h>

SEL **sel_getUid**(STR *aName*)
STR **sel_getName**(SEL *aSelector*)

DESCRIPTION

The first function, **sel_getUid()**, returns the unsigned integer that's used at run time to identify the *aName* method. Whenever possible, you should use the **@selector()** directive to ask the compiler, rather than the run-time system, to provide the selector for a method. This function should be used only if the name isn't known at compile time.

The second function, **sel_getName()**, is the inverse of the first. It returns the name that was mapped to *aSelector.*

RETURN

sel_getUid() returns the selector for the *aName* method, or 0 if there is no known method with that name. **sel_getName**() returns a character string with the name of the method identified by the *aSelector* selector. If *aSelector* isn't a valid selector, a NULL pointer is returned.

sel_isMapped()

SUMMARY Determine whether a selector is valid

LIBRARY libsys_s.a

SYNOPSIS

#import <objc/objc.h>

BOOL **sel_isMapped**(SEL *aSelector*)

RETURN

sel_isMapped() returns YES if *aSelector* is a valid selector (is currently mapped to a method implementation) or could possibly be one (because it lies within the same range as valid selectors); otherwise it returns NO.

Because all of a program's selectors are guaranteed to be mapped at start-up, this function has little real use. It's included here for reasons of backward compatibility only.

_alloc(), _dealloc(), _realloc(), _copy(), _zoneAlloc(), _zoneRealloc(), _zoneCopy(), _error()

SUMMARY Set functions used by the run-time system

LIBRARY libsys_s.a

SYNOPSIS

#import <objc/objc-runtime.h>

id (***_alloc**)(Class *aClass*, unsigned int *indexedIvarBytes*)
id (***_dealloc**)(Object **anObject*)
id (***_realloc**)(Object **anObject*, unsigned int *numBytes*)
id (***_copy**)(Object **anObject*, unsigned int *indexedIvarBytes*)
id (***_zoneAlloc**)(Class *aClass*, unsigned int *indexedIvarBytes*, NXZone **zone*)
id (***_zoneRealloc**)(Object **anObject*, unsigned int *numBytes*, NXZone **zone*)
id (***_zoneCopy**)(Object **anObject*, unsigned int *indexedIvarBytes*, NXZone **zone*)
void (***_error**)(Object **anObject*, char **format*, va_list *ap*)

DESCRIPTION

These variables point to the functions that the run-time system uses to manage memory and handle errors. By reassigning a variable, a function can be replaced with another of the same type. The example below shows a temporary reassignment of the **_zoneAlloc** function:

```
id (*theFunction)();
theFunction = _zoneAlloc;
_zoneAlloc = someOtherFunction;
/*
 * code that calls the class_createInstanceFromZone() function,
 * or sends alloc and allocFromZone: messages, goes here
 */
_zoneAlloc = theFunction;
```

- **_alloc** points to the function, called through **class_createInstance()**, used to allocate memory for new instances, and **_zoneAlloc** points to the function, called through **class_createInstanceFromZone()**, used to allocate the memory for a new instance from a specified *zone*.
- **_dealloc** points to the function, called through **object_dispose()**, used to free instances.
- **_realloc** points to the function, called through **object_realloc()**, used to reallocate memory for an object, and **_zoneRealloc** points to the function, called through **object_reallocFromZone()**, used to reallocate memory from a specified *zone*.
- **_copy** points to the function, called through **object_copy()**, used to create an exact copy of an object, and **_zoneCopy** points to the function, called through **object_copyFromZone()**, used to create the copy from memory in the specified *zone*.
- **_error** points to the function that the run-time system calls in response to an error. By default, it prints formatted error messages to the standard error stream and calls **abort()** to produce a core file.

_copy → See _alloc

_dealloc → See _alloc

_error → See _alloc

_realloc → See _alloc

_zoneAlloc → See _alloc

_zoneCopy → See _alloc

_zoneRealloc → See _alloc

Chapter 4
PostScript Operators

This chapter contains detailed descriptions of NeXT's extensions to the Display PostScript system. It also lists the standard PostScript operators that have different or additional effects in the NeXT implementation. For information on the standard PostScript language operators, see the *PostScript Language Reference Manual*. See the *Extensions for the Display PostScript System* manual for details on the operators Adobe Systems Incorporated added for the Display PostScript system. Information on these and other references for the PostScript language is listed in "Suggested Reading" in the *NeXT Technical Summaries* manual.

The operators marked "internal" shouldn't be used in applications based on the Application Kit since your use of them will conflict with the Kit's.

This chapter presents the operators in alphabetical order and uses the same format as that of the operator descriptions in the *PostScript Language Reference Manual*. The *Technical Summaries* manual provides a complete summary of all PostScript operators, organized into groups of related operators. Chapter 3, "C Functions," describes the C interface to the operators listed in this chapter.

adjustcursor

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

pixelswide pixelshigh bits/sample matrix $proc_0$ *[...* $proc_n$*] multiproc ncolors*
alphaimage –

Renders an image whose samples each contain one, three, or four color components plus an alpha component. (Most programmers should use **NXImageBitmap()** instead of **alphaimage**.)

This operator is modeled on the **colorimage** operator as described in *PostScript Language Color Extensions* (see "Suggested Reading" in the *NeXT Technical Summaries* manual). It differs from **colorimage** in that it assumes an alpha component in addition to the color components for each sample.

The sampled image is a rectangular array of *pixelswide***pixelshigh* pixels. For each pixel, there must be *ncolors* color components and one alpha component. The only valid possibilities for *ncolors* are 1 (gray scale), 3 (RGB), and 4 (CMYK). Each color and alpha component is represented by *bits/sample* bits. Each color component is premultiplied; that is, it's the result of the prior multiplication of the color contribution and the corresponding alpha value. (See "Premultiplication" in the *Concepts* manual for more information.)

alphaimage calls its procedure operand(s) repeatedly to get the color and alpha values to be rendered. See *PostScript Language Color Extensions* for a discussion of the data formats that these procedures must return.

multiproc is a boolean value referring to whether the color and alpha components are each supplied separately (*multiproc* is *true*) or interleaved (*multiproc* is *false*). In the single-procedure form (*multiproc* is *false*), the samples are GA (the gray and alpha components), RGBA (RGB components plus an alpha component), or CMYKA (CMYK components plus an alpha component). In the multiple-procedure form (*multiproc* is *true*), the alpha procedure is last ($proc_{ncolors}$); for example, for *ncolors*=1, this operator has the form:

pixelswide pixelshigh bits/sample matrix dataproc alphaproc true 1
alphaimage –

ERRORS
invalidid, **limitcheck**, **rangecheck**, **stackunderflow**, **typecheck**, **undefined**, **undefinedresult**

banddevice

matrix width height proc **banddevice** – % undefined

This standard PostScript operator is not defined in the NeXT implementation of the Display PostScript system.

basetocurrent

x y **basetocurrent** *x' y'*

Converts (*x*, *y*) 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

x y **basetoscreen** *x' y'*

Converts (*x*, *y*) 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

– **buttondown** *bool*

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**

cleardictstack

– **cleardictstack** –

Returns the dictionary stack to its initial state, in which it contains only **systemdict**, **shareddict**, and **userdict**. **cleardictstack** should be used instead of counting the number of dictionaries to pop off—that is, instead of

```
    { countdictstack 2 ge { exit } end } loop
```

Note: Adobe has recently added this operator to the Display PostScript system. This entry will be removed when **cleardictstack** is documented in Adobe's manuals.

ERRORS
  dictstackunderflow

cleartrackingrect

trectnum gstate **cleartrackingrect** –

Clears the tracking rectangle with the number *trectnum*, as set by **settrackingrect**, in the device referred to by *gstate*. If no such rectangle exists, the **invalidid** error is executed. If *gstate* is null, the current graphics state is assumed.

ERRORS
  invalidid, **stackunderflow**, **typecheck**

SEE ALSO
  settrackingrect

composite *src_x src_y width height srcgstate $dest_x$ $dest_y$ 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 *srcgstate* or the current graphics state doesn't refer to a window device, the **invalidid** error is executed.) The remaining operands define the shape that contains the source and destination pixels and the locations of that shape in the current coordinate system of the respective graphics states. The result of an operation on a source and destination pixel replaces the destination pixel.

The rectangle specified by *src_x*, *src_y*, *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; for more information, see the general rule given in the *Concepts* manual, under "Imaging Conventions."

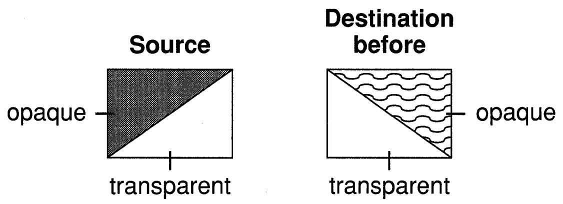
There's one destination pixel for each pixel in the source. The source and destination images have the same size, shape, and orientation. (Even if the axes have a different orientation in the source and destination graphics states, the images will not; **composite** will not rotate images.) In screen coordinates, the difference between *src_x* and *$dest_x$* —both truncated **float** values—is the x displacement between all source and destination pixels; *src_y* and *$dest_y$* similarly determine the y displacement.

The source image is clipped to the frame rectangle of the window in the source graphics state, and the destination image is clipped to the frame rectangle and clipping path of the window in the current graphics state.

op specifies the compositing operation. The choices for *op* and the result of each operation are given in Figure 4-1 on the following page. For a detailed explanation of each operator, see "Types of Compositing Operations" in the *Concepts* manual.

ERRORS
invalidid, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO
compositerect, **setalpha**, **setgray**, **sethsbcolor**, **setrgbcolor**



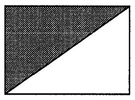
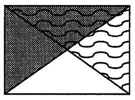
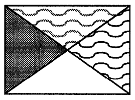
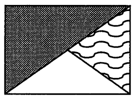
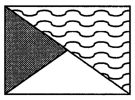
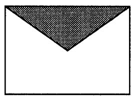
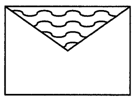
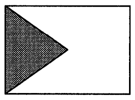
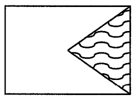
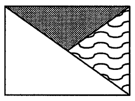
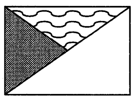
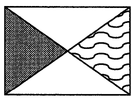
Operation	Destination after	
Copy		Source image.
Clear		Transparent.
PlusD		Sum of source and destination images, with color values approaching 0 as a limit.
PlusL		Sum of source and destination images, with color values approaching 1 as a limit. (PlusL is not implemented for the MegaPixel Display.)
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.
Datop		Destination image wherever both images are opaque, source image wherever source image is opaque but destination image is transparent, and transparent elsewhere.
Xor		Source image wherever source image is opaque but destination image is transparent, destination image wherever destination image is opaque but source image is transparent, and transparent elsewhere.

Figure 4-1. Compositing Operations

compositerect *$dest_x$ $dest_y$ 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; *src_x*, *src_y*, *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; for more information, see the general rule given in the *Concepts* manual, under "Imaging Conventions." 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**, **setgray**, **sethsbcolor**, **setrgbcolor**

copypage – **copypage** – % different in the NeXT implementation

This standard PostScript operator has no effect in the NeXT implementation of the Display PostScript system.

ERRORS
none

SEE ALSO
erasepage, **showpage**

countframebuffers

– **countframebuffers** *count*

Returns the number of frame buffers that the Window Server is actually using.

ERRORS
stackoverflow

SEE ALSO
framebuffer

countscreenlist

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

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 – **currentactiveapp** *context* % internal

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 – **currentalpha** *coverage*

Returns the coverage parameter of the current graphics state.

ERRORS
  none

SEE ALSO
  composite, **setalpha**

currentdefaultdepthlimit
– **currentdefaultdepthlimit** *depth* % internal

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 *window* **currentdeviceinfo** *min max bool*

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 *bool* is a boolean value indicating whether the device is a color device.

ERRORS
  invalidid, **stackunderflow**, **typecheck**

currenteventmask *window* **currenteventmask** *mask* % internal

Returns the current Window Server-level event mask for the specified window. For windows created by the Application Kit, this mask may allow additional event types beyond those requested by the application.

Normally you should use the Window object's **eventMask** method instead of the **currenteventmask** operator. Use this operator only if you're bypassing the Application Kit.

ERRORS
invalidid, **stackunderflow**, **typecheck**

SEE ALSO
seteventmask

currentmouse *window* **currentmouse** *x y* % internal

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.

Normally you should use the Window object's **getMouseLocation:** method instead of the **currentmouse** operator. Use this operator only if you're bypassing the Application Kit.

ERRORS
invalidid, **stackunderflow**, **typecheck**

SEE ALSO
basetocurrent, **basetoscreen**, **buttondown**, **rightbuttondown**, **rightstilldown**, **setmouse**, **stilldown**

currentowner *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**

currentrusage

– **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 UNIX 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
nivcsw	int	Number of involuntary context switches

currenttobase

x y **currenttobase** *x' y'*

Converts (*x*, *y*) 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

x y **currenttoscreen** *x' y'*

Converts (*x*, *y*) 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 – **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
context **currentwaitcursorenabled** *bool*

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 – **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 *window* **currentwindowalpha** *state*

Returns an integer indicating whether the Window Server is currently storing alpha values for the specified window. Possible *state* 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

window **currentwindowbounds** *x y width height*

Returns the location and size of the window in screen coordinates. You can pass 0 for *window* to determine the size of the entire workspace, that is, the smallest rectangle that encloses all active screens.

x and *y* will be integers in the range from -2^{15} to $2^{15}-1$; *width* and *height* will be integers in the range from 0 to 10000.

Normally you should use the Window object's **getFrame:** method instead of this operator (or the Application object's **getScreenSize:** method, for the size of the screen). Use this operator only if you're bypassing the Application Kit.

ERRORS
invalidid, **stackunderflow**, **typecheck**

SEE ALSO
movewindow, **placewindow**

currentwindowdepth

window **currentwindowdepth** *depth* % internal

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

window **currentwindowdepthlimit** *depth* % internal

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

window **currentwindowdict** *dict* % internal

Returns the specified window's dictionary. Every window created by the Application Kit has a dictionary associated with it. Since the Application Kit uses this dictionary internally, direct manipulation of it will probably cause errors. Avoid calling this operator.

ERRORS
invalidid, **stackunderflow**, **typecheck**

SEE ALSO
setwindowdict

currentwindowlevel

window **currentwindowlevel** *level*

Returns *window*'s tier. Executes the **invalidid** error if *window* doesn't exist.

ERRORS
invalidid, **stackunderflow**, **typecheck**

SEE ALSO
setwindowlevel

currentwriteblock

– **currentwriteblock** *bool*

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 *bool* is *true*, the Server waits until the buffer can accept more data. If *bool* is *false*, the Server discards data that can't be accepted immediately.

SEE ALSO
setwriteblock

dissolve *src_x src_y width height srcgstate $dest_x$ $dest_y$ 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 *dumplevel window* **dumpwindow** – % internal

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 *dumplevel context* **dumpwindows** – % internal

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

– **erasepage** – % different in the NeXT implementation

This standard PostScript operator has the following effect in the NeXT implementation of the Display PostScript system: It erases the entire window to opaque white.

ERRORS
invalidid

SEE ALSO
copypage, **showpage**

findwindow

x y place otherwindow **findwindow** *x' y' window bool*

findwindow starts from a given position in the screen list and searches for the uppermost window below the 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

– **flushgraphics** –

Flushes to the screen all drawing done in the current buffered window. If the current window is retained or nonretained, **flushgraphics** has no effect.

Normally you should use the Window object's **flushWindow** method instead of this operator. Use this operator only if you're bypassing the Application Kit.

ERRORS
invalidid, **stackunderflow**, **typecheck**

framebuffer

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, NX_TwentyFourBitRGB).

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

– **frontwindow** *window* % internal

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

– **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

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**

initgraphics

– **initgraphics** – % different in the NeXT implementation

This standard PostScript operator has these additional effects in the NeXT implementation of the Display PostScript system:

- Sets the coverage parameter in the current window's graphics state to 1 (opaque)
- Turns off instance drawing

ERRORS
none

SEE ALSO
hideinstance, **newinstance**, **setalpha**, **setinstance**

machportdevice

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. (See **/usr/include/windowserver/printmessage.h** for the structure used in the print message.)

width and *height* are integers that determine the number of device pixels for the page. *bbox* is an array of integers in the form [*llx lly urx ury*]. The array specifies the lower left and upper right corners of the rectangle in the device raster to use as the boundary of the imageable area. (For the common case where the entire raster is imageable, *bbox* may be expressed as a zero-length array, [ ], which **machportdevice** interprets as [0 0 *width height*].) **machportdevice** requires the bounding box array *bbox* to be well formed and within the device pixel bounds of [0 0 *width height*]; otherwise, 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. *pixelencoding* is a dictionary describing the format for the image data rendered by the Window Server. It should contain these entries:

Key	**Type**	**Semantics**
samplesPerPixel	integer	Currently must be 1
bitsPerSample	integer	Currently 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** in the pixel-encoding dictionary should be one of the following values, predefined in **nextdict**.

Name	**Value**	**Description**
NX_OneIsBlackColorSpace	0	Monochromatic, high sample value is black.
NX_OneIsWhiteColorSpace	1	Monochromatic, high sample value is white.
NX_RgbColorSpace	2	RGB, (1,1,1) is white.
NX_CmykColorSpace	5	CMYK, (0,0,0,0) is white.

The current implementation of **machportdevice** supports only the following combinations of **colorSpace** and **bitsPerSample**:

colorSpace	**bitsPerSample**
NX_OneIsBlackColorSpace	1
NX_OneIsWhiteColorSpace	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

portname is resolved from the nameserver on *hostname* by calling **netname_look_up**(). This occurs during the execution of **machportdevice**—not for each page—so be sure that the receiving port has been checked in using **netname_check_in**() prior to executing **machportdevice**. If the portname isn't checked in on the given host, a **rangecheck** results.

If *hostname* is of length 0, the local host is assumed. If it is equal to '*', a broadcast lookup is performed by **netname_look_up()**. Note, however, that sending large pages to remote hosts causes considerable network traffic, while sending large pages to the local host won't require any copying of physical memory.

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. (In general, an effort is made to gracefully ignore all LaserWriter-specific commands, which are listed in Appendix D of the *PostScript Language Reference Manual.*)

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, we've chosen to put the user space origin 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 *x y window* **movewindow** – % internal

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.

Normally you should use the Window object's **moveTo::** method instead of this operator. Use this operator only if you're bypassing the Application Kit.

ERRORS

invalidid, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO

currentwindowbounds, **placewindow**

newinstance – **newinstance** –

Removes any instance drawing from the current window.

ERRORS

invalidid

SEE ALSO

hideinstance, **setinstance**

nextrelease – **nextrelease** *string*

Returns version information about this release of the NeXT Window Server.

ERRORS

stackoverflow

SEE ALSO

osname, **ostype**

NextStepEncoding

– **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. See Chapter 6 of the *NeXT Technical Summaries* manual for a table listing the character names and corresponding characters of this vector.

ERRORS
stackoverflow

SEE ALSO
StandardEncodingVector

obscurecursor

– **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 *place otherwindow window* **orderwindow** – % internal

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 the screen list.
- 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.

Since the workspace is a window in the screen list, **Below 0** will make the specified window disappear behind all other windows, including the workspace. To place a window just above the workspace window, you can use **Above workspaceWindow**. (**workspaceWindow** is a PostScript name whose value is the window number of the workspace window.)

Note: orderwindow doesn't change which window is the current window.

Normally you should use the Window object's **orderWindow:relativeTo:** method instead of the **orderwindow** operator. Use this operator only if you're bypassing the Application Kit.

ERRORS
invalidid, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO
frontwindow

osname

– **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. See the *PostScript Language Reference Manual* for more information on **statusdict**. **osname** can be executed as follows:

```
statusdict /osname get exec
```

The NeXT version of the Window Server returns the string:

```
(NeXT Mach)
```

ERRORS
none

SEE ALSO
nextrelease, ostype

ostype

– **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. See the *PostScript Language Reference Manual* for more information on **statusdict**. **ostype** can be executed as follows:

```
statusdict /ostype get exec
```

The NeXT version of the Window Server returns the number 3 to indicate the operating system is a variant of UNIX.

ERRORS
none

SEE ALSO
nextrelease, osname

placewindow *x y width height window* **placewindow** – % internal

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 4-2). Any other areas of the newly positioned window are filled with the window's exposure color (see **setexposurecolor**).

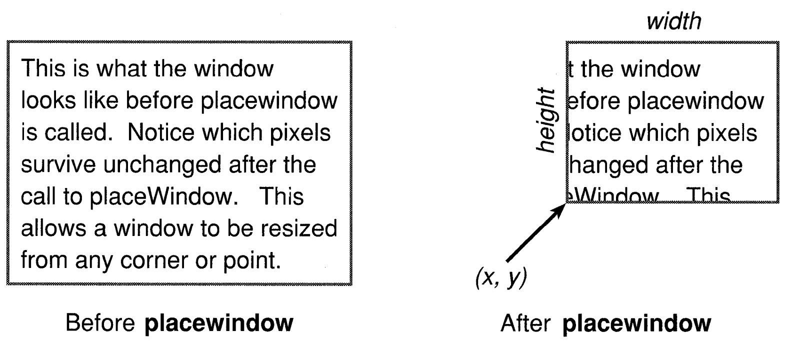

This is what the window looks like before placewindow is called. Notice which pixels survive unchanged after the call to placeWindow. This allows a window to be resized from any corner or point.
Before placewindow
width
height
t the window
efore placewindow
otice which pixels
hanged after the
Window. This
(x, y)
After placewindow

Figure 4-2. **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.

Normally you should use the Window object's **placeWindow:** method instead of the **placewindow** operator. The **placeWindow:** method reestablishes the window's transformation matrix and clipping path for you. Use the **placewindow** operator only if you're bypassing the Application Kit.

ERRORS
invalidid, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO
currentwindowbounds, **movewindow**, **setexposurecolor**

playsound *soundname priority* **playsound** –

Plays the sound *soundname*. The Window Server searches for a standard NeXT 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 *type x y time flags window subType misc0 misc1 context* **posteventbycontext** *bool*

Posts an event to the specified context. The nine parameters preceding the context parameter coincide with the NXEvent structure members (see **dpsclient/events.h**). 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 might use this operator to post an application-defined event to your own application. Use Mach messaging to communicate between applications.

ERRORS
stackunderflow, **typecheck**

readimage *x y width height proc*$_0$ *[... proc*$_{n-1}$*] string bool* **readimage** –

Reads the pixels that make up a 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. In any case, the pixels included in the image are determined by the rules given in the *Concepts* manual, under "Imaging Conventions."

You would 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**

renderbands

proc **renderbands** – % undefined

This standard PostScript operator is not defined in the NeXT implementation of the Display PostScript system.

revealcursor

– **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

– **rightbuttondown** *bool*

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

eventnum **rightstilldown** *bool*

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 *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 *x y* **screentobase** *x' y'*

Converts (*x*, *y*) 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

x y **screentocurrent** *x' y'*

Converts (*x*, *y*) 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

context **setactiveapp** – % internal

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

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**, **setgray**, **sethsbcolor**, **setrgbcolor**

setautofill *bool window* **setautofill** –

Applies only to nonretained windows; sets the autofill property of *window* to *true* or *false*. 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 *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

depth **setdefaultdepthlimit** – % internal

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**
NX_TwoBitGray	1 spp, 2bps, 2bpp, planar
NX_EightBitGray	1 spp, 8bps, 8bpp, planar
NX_TwelveBitRGB	3 spp, 4bps, 16bpp, interleaved
NX_TwentyFourBitRGB	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 NX_TwoBitGray and 16 for NX_EightBitGray). Interleaved configurations already account for an alpha sample whether or not it's present; thus, the number of bits per pixel for NX_TwelveBitRGB and NX_TwentyFourBitRGB depths remains unchanged.

The constant NX_DefaultDepth is also available. If *depth* is NX_DefaultDepth, 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 *mask window* **seteventmask** – % internal

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	**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

Normally you should use the Window object's **setEventMask:** method instead of the **seteventmask** operator. Use this operator only if you're bypassing the Application Kit.

ERRORS
invalidid, **stackunderflow**, **typecheck**

SEE ALSO
currenteventmask

setexposurecolor

– **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

bool **setflushexposures** – % internal

Sets whether window-exposed and screen-changed subevents are flushed to clients. If *bool* is *false*, no window-exposed or screen-changed events are flushed to the client until **setflushexposures** is executed with *bool* equal to *true*. By default, window-exposed and screen-changed events are flushed to clients.

ERRORS
invalidid, **stackunderflow**, **typecheck**

setinstance

bool **setinstance** –

Sets the instance-drawing mode in the current graphics state on (if *bool* is *true*) or off (if *bool* is *false*).

ERRORS
stackunderflow, **typecheck**

SEE ALSO
hideinstance, **newinstance**

setmouse

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

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**

setpattern

patternname **setpattern** –

Sets the current pattern parameter in the graphics state to *patternname*. The pattern overrides the current color in the graphics state. Pattern drawing is automatically disabled when any other operator sets the current color in the graphics state (for example, **setgray**, **setrgbcolor**, or **setalpha**). This operator should be used for drawing user interface elements that can't be drawn in one of the four pure gray values. By using a dither pattern rather than an intermediate shade of gray, you avoid having windows promoted to greater depths on the basis of standard user-interface features. For example, Scroller uses a pattern to draw the gray shade behind the knob.

Only the following three patterns (defined in **nextdict**) are permitted:

NX_MediumGrayPattern	(50% dither of .333 and .666 gray)
NX_LightGrayPattern	(50% dither of .666 and 1.0 gray)
NX_DarkGrayPattern	(50% dither of 0 and .333 gray)

The **setpattern** operator only works if the current device is a window; if it's something other than a window (such as a printer, as set by **machportdevice**) an error occurs.

This operator will be superseded by PostScript Level 2's **setpattern** operator. (The above patterns will continue to work, however.)

ERRORS
invalidid, **stackunderflow**

SEE ALSO
adjustcursor, **basetocurrent**, **currentmouse**, **screentocurrent**

setsendexposed

bool window **setsendexposed** – % internal

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**

settrackingrect

x y width height leftbool rightbool insidebool userdata trectnum gstate
settrackingrect –

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.) The application will thereafter receive mouse-exited and mouse-entered events as the cursor leaves and reenters the visible portion of the tracking rectangle. Any number of tracking rectangles may be set in a single window.

Note: You normally use the Window class's **setTrackingRect:inside:owner:tag:left:right:** method for general cursor tracking. To track the cursor and change its image based on its location, you'd normally use the Window class's cursor management methods such as **addCursorRect:cursor:forView:**.

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. In the event record for a mouse-exited or mouse-entered event generated as a result of this call to **settrackingrect**, the **data** component will contain *trectnum* along with the event number of the last mouse-down event.

userdata is also an arbitrary integer that you assign to a tracking rectangle. However, 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.

The tracking rectangle will remain in effect until **cleartrackingrect** is called, or until another tracking rectangle with the same *trectnum* is set.

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.

ERRORS
invalidid, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO
cleartrackingrect

setwaitcursorenabled *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

depth window **setwindowdepthlimit** – % internal

Sets the depth limit of *window* to *depth*. These depths are defined in **nextdict**:

Depth	**Meaning**
NX_TwoBitGray	1 spp, 2bps, 2bpp, planar
NX_EightBitGray	1 spp, 8bps, 8bpp, planar
NX_TwelveBitRGB	3 spp, 4bps, 16bpp, interleaved
NX_TwentyFourBitRGB	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 NX_TwoBitGray and 16 for NX_EightBitGray). Interleaved configurations already account for an alpha sample whether or not it's present; thus, the number of bits per pixel for NX_TwelveBitRGB and NX_TwentyFourBitRGB depths remains unchanged.

Another constant, NX_DefaultDepth, is defined as the default depth limit in the Window Server's current context. If *depth* is NX_DefaultDepth, 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 *dict window* **setwindowdict** – % internal

Sets the dictionary for *window* to *dict*. This is usually done by the Application Kit.

Every window created by the Application Kit has a dictionary associated with it. Since the Application Kit uses this dictionary internally, direct manipulation of it will probably cause errors. Avoid using this operator.

ERRORS
invalidid, stackunderflow, typecheck

SEE ALSO
currentwindowdict

setwindowlevel *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 *type window* **setwindowtype** –

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 *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 – **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 – **showpage** – % different in the NeXT implementation

This standard PostScript operator has no effect if the current device is a window.

ERRORS
none

SEE ALSO
copypage, **erasepage**

sizeimage *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***
NX_TwoBitGray	1	2	*true*
NX_EightBitGray	1	8	*true*
NX_TwelveBitRGB	3	4	*false*
NX_TwentyFourBitRGB	3	8	*false*

ERRORS
stackunderflow, **typecheck**

SEE ALSO
alphaimage, **readimage**

stilldown *eventnum* **stilldown** *bool*

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 *window* **termwindow** – % internal

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.

Normally you should use the Window object's **close** method instead of the **termwindow** operator. Use this operator only if you're bypassing the Application Kit.

ERRORS
invalidid, **stackunderflow**, **typecheck**

SEE ALSO
window, **windowdevice**, **windowdeviceround**

window *x y width height type* **window** *window* % internal

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**.

Normally you should use the Window object's **newContent:style:backing:buttonMask:defer:** method instead of the **window** operator. Use this operator only if you're bypassing the Application Kit.

ERRORS
invalidid, **rangecheck**, **stackunderflow**, **typecheck**

SEE ALSO
setexposurecolor, **termwindow**, **windowdeviceround**

windowdevice

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 NeXTstep 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

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, approximately 1/92 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.

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

windowdevice

windowlist *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**

Chapter 5
Data Formats

Chapter 5
Data Formats

To make it easier for applications to share information, the NeXTstep pasteboard supports a small number of standard data formats. Each format, or type, is identified by a global variable:

Variable Name	**Type Description**
NXAsciiPboardType	Plain ASCII text
NXPostScriptPboardType	Encapsulated PostScript code (EPS)
NXTIFFPboardType	Tag Image File Format (TIFF)
NXRTFPboardType	Rich Text Format (RTF)
NXSoundPboardType	The Sound object's pasteboard type
NXFilenamePboardType	ASCII text designating a file name
NXTabularTextPboardType	Tab-separated fields of ASCII text
NXFontPboardType	Font and character information
NXRulerPboardType	Paragraph formatting information

Data in other formats can also be placed in the pasteboard. However, the sending and receiving applications must both agree on the structure of the format, its name, and how to interpret it. Other formats may be adopted as standards in the future.

Each of the standard formats is discussed below. In most cases, the discussion is short and consists only of a reference to the primary source document for the format. In some cases, more information is given on modifications to or interpretations of the format in the NeXTstep environment.

NXAsciiPboardType

Text in this format consists only of characters from the ASCII character set as extended by NeXTstep encoding. None of the characters is given a special interpretation (in contrast to NXTabularTextPboardType and NXFilenamePboardType, for example). Standard ASCII is documented on-line in **/usr/pub/ascii** and the **ascii**(7) manual page. NeXTstep encoding is documented in Chapter 6 of the *NeXT Technical Summaries* manual.

NXPostScriptPboardType

This type is defined as PostScript code in the Encapsulated PostScript Files format (EPS). The PostScript language is documented by Adobe Systems Incorporated, principally in the *PostScript Language Reference Manual*, published by Addison-Wesley. EPS conventions are documented in *Encapsulated PostScript Files Specification*, by Adobe Systems Incorporated.

NXTIFFPboardType

This type is for image data in Tag Image File Format (TIFF). TIFF is documented in *Tag Image File Format Specification*, by Aldus Corporation and Microsoft Corporation.

TIFF support in the current NeXTstep release follows version 5.0 of the TIFF standard and is based on version 2.2 of Sam Leffler's freely distributed TIFF library. This library provides a good set of routines for dealing with TIFF files that conform to the 5.0 specification.

NeXTstep TIFF support is embodied in the NXBitmapImageRep class and the command-line program **tiffutil**. See "NXBitmapImageRep" in Chapter 2, "Class Specifications" and the **tiffutil** manual page for more information.

Unsupported Fields

In the current release, some fields—principally those having to do with response curves—will be read correctly but ignored when imaging the data. Color palettes are not supported except when the palette entries are 8 bits and the stored colors are 24 bits. These files will be read correctly and converted to 24-bit images on the fly.

The Matte Field

The 5.0 TIFF specification has been extended to include a Matte field (tag 32995), which indicates the presence or absence of a coverage component (alpha) in the data. This field is a SHORT with a value of 1 or 0. A value of 1 indicates that a coverage component is present and that the color components are premultiplied by the alpha values. The coverage component follows the color components in the data. The absence of this field or a value of 0 indicates that no coverage component is present; the image is opaque.

TIFF files generated by release 1.0 of NeXTstep did not contain a Matte field. Instead, to indicate the presence of a coverage component, the value of the SamplesPerPixel field was set to 2 and the value of the PhotometricInterpretation field was set to 5. Release 2.0 software recognizes these files as containing alpha despite the lack of a Matte field. Thus all TIFF files generated by 1.0 software will be interpreted correctly.

Multiple Images

Multiple forms of an image can now be stored in the same file—that is, under the same TIFF header. "Multiple forms" might mean the same image at different resolutions (for example, 72dpi and 400dpi) and at different bit depths or colors (for example, 2 bits per sample on a gray scale and 4 bits per sample RGB).

This feature is useful when you want to create color icons for an application and its documents. It's best to create both gray scale and color versions of the icons and store them in the same section of the ICON segment. Both versions of the icon would be created at 72 dpi and would be 48 pixels wide by 48 pixels high. The gray-scale version would have two components (gray and alpha), with each component stored at 2 bits. The color version would have 4 components (red, green, blue, and alpha) and each component would be 4 bits deep. (It's recommended that application and document icons be stored at 4 bits per sample, not 8.)

Compression

NeXTstep software can both read and write compressed TIFF images. The Compression field in a TIFF file can have any of the following values:

Compression Value	Compression Type
1	No compression
5	LZW (Lempel-Ziv & Welch) compression
32773	PackBits compression
32865	JPEG compression

JPEG compression can be used only for images that have a depth of at least 4 bits per sample.

NXRTFPboardType

This is the pasteboard type for "rich text," text that follows the conventions of the Rich Text Format®, as described in *Rich Text Format Specification* by Microsoft Corporation.

To this specification, NeXT has added a control word to indicate how the user selected the text before copying it to the pasteboard. The control word is

\smartcopy<*num*>

where <*num*> can be 1 or 0. A value of 1 indicates that the user made the selection by double-clicking a word, or double-clicking and dragging over a group of words. The range of text in the pasteboard will be delimited by a word boundary on either side. The pasting application can use this information to correctly adjust the spacing around the word or words that are pasted.

NXSoundPboardType

This format is defined by the SNDSoundStruct structure in the header file **sound/soundstruct.h**. The structure and the methods for writing sound data to and reading it from the pasteboard are discussed in more detail in *Sound, Music, and Signal Processing*.

NXFilenamePboardType

This format is a list of tab-separated file names (or pathnames), terminated by a null character ('\0').

NXTabularTextPboardType

This format is ASCII text where tabs (ASCII 0x09) and returns or newlines (ASCII 0x0D) are interpreted as separators between text fields. In a matrix, tabs separate columns and returns separate rows. The text is null-terminated.

NXFontPboardType

This format is used in the font pasteboard to record character properties that are copied and pasted using the Copy Font and Paste Font commands. It consists of RTF control words from the "Font Table" and "Character Formatting Properties" groups.

The following is an example of character data in this format:

```
{\rtf1\ansi{\fonttbl\f0\froman Times;}
\f0\b0\i\ul0\fs48}
```

The first two control words, **\rtf1** and **\ansi**, announce that the information enclosed within the outer braces is RTF version 1 in ANSI character encoding. These two control words, or their equivalent, are required by RTF conventions.

The group within the inner braces defines a font table, here with a single entry specifying font 0 to be Times-Roman. The font is then specified as Times-Roman (font 0), not bold, Oblique (italic), not underlined, and having a font size of 24 points (48 half points).

Among the fonts that can be specified in a font table are these:

```
\fmodern Courier;
\fswiss Helvetica;
\fmodern Ohlfs;
\ftech Symbol;
\froman Times;
```

Several synonyms are recognized for the Times-Roman font. Usually it's written as "Times" or "Times-Roman".

If the font pasteboard contains RTF control words that don't belong to the "Font Table" or "Character Formatting Properties" groups, they should be ignored. If control words specify more than one value for a font characteristic, the last value specified should be used when pasting.

NXRulerPboardType

This format is used in the ruler pasteboard to capture information about how a paragraph is formatted. It consists of RTF control words from the "Paragraph Formatting Properties" group.

The following is an example of this type:

```
{\rtf1\ansi
\pard\ql\tx1252\tx2716\tx4148\tx5592\tx7004\tx11520
\fi-540\li1260}
```

The first two control words are required by RTF conventions, as explained under "NXFontPboardType" above. The next control word, **\pard**, resets the paragraph format to the default. The paragraph is then specified to be left-aligned and a series of six tabs are set. Next, the indentation of the first line is specified and, finally, the left indent. (The example is for a paragraph with a hanging indent.)

If the ruler pasteboard contains RTF control words that aren't in the "Paragraph Formatting Properties" group, they should be ignored. If it includes control words that first set then reset a paragraph property, the final specification should be the one that's used.

Index

NeXT Computer, Inc.
900 Chesapeake Drive
Redwood City, CA 94063

Printed in U.S.A.
2909.00
12/90