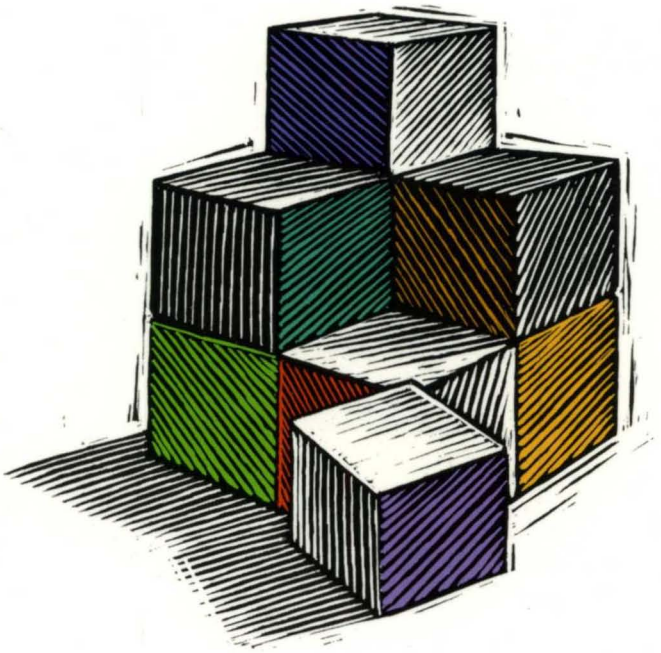
GENERAL REFERENCE
VOLUME 1



NEXTSTEP™

Object-Oriented Software

NeXTSTEP™ GENERAL REFERENCE

Volume 1

NeXTSTEP Developer's Library
NeXT Computer, Inc.

Release 3

Addison-Wesley Publishing Company
Reading, Massachusetts · Menlo Park, California · New York · Don Mills, Ontario
Wokingham, England · Amsterdam · Bonn · Sydney · Singapore · Tokyo · Madrid
San Juan · Paris · Seoul · Milan · Mexico City · Taipei

This manual describes NeXTSTEP Release 3.

Written by NeXT Publications.

This manual was designed, written, and produced on NeXT computers. Proofs were printed on a NeXT 400 dpi Laser Printer and NeXT Color Printer. Final pages were transferred directly from a NeXT optical disk to film using NeXT computers and an electronic imagesetter.

3 4 5 6 7 8 9 10–CRS–96959493
Third printing, November 1993

ISBN 0-201-62220-3

Contents

Volume 1:

Volume 2:

Introduction

This manual describes the application programming interface (API) for the NeXTSTEP™ development environment. It's part of a collection of manuals called the *NeXTSTEP Developer's Library*, which offer assistance to developers creating applications for NeXTSTEP computers. Some of the other manuals in the library are listed on the back cover.

The two volumes of the *General Reference* provide detailed descriptions of all the NeXTSTEP software kits and of all the classes, functions, operators, and other programming elements that make up the API. The first volume covers the root Object class, Application Kit™, and other common classes and functions. The second volume covers more specialized kits, like Database Kit™, Phone Kit™, and 3D Graphics Kit™. Most programmers will use the Application Kit and one or more of the other kits, depending on the kind of application they're developing.

The information in these volumes is supplemented by on-line release notes (in the **/NextLibrary/Documentation/NextDev/ReleaseNotes** directory) that you can access through the Digital Librarian application. The release notes provide last-minute information about the latest release of the software.

The Mach operating system is documented in another *Developer's Library* manual, *NeXTSTEP Operating System Software*. For the most part, you don't have to be familiar with Mach to use the Application Kit and other software documented here.

However, this manual does assume that you're familiar with the NeXTSTEP user interface, with the C programming language, and with the Objective C extensions to C. Objective C is documented in *NeXTSTEP Object-Oriented Programming and Objective C*. The user interface is described and explained in *NeXTSTEP User Interface Guidelines*.

Using Documented API

The API described in this manual provides all the functionality you need to make full use of NeXTSTEP software. If you have questions about using the API, this documentation and the NeXT Technical Support Department can help you use it correctly. If a feature in the API doesn't work as described, it's considered a bug which NeXT will work to fix. If API features change in future releases, these changes will be described in on-line release notes and printed documentation.

Undocumented features are not part of the API. If you use undocumented features, you run several risks. First, your application may be unreliable, because undocumented features won't work the way you expect them to in all cases. Second, NeXT Technical Support can't provide full assistance in fixing problems that arise, other than to recommend that you use documented API. Finally, your application may be incompatible with future releases, since undocumented features can and will change without notice.

Precompiled Header Files

Throughout this manual, you'll find cross references to the header files where NeXTSTEP API is declared. All these header files are located in subdirectories of the **/NextDeveloper/Headers** directory.

When programming, you typically import the header files that declare the API you're using. For example, to use the NXPhoneCall class, you'd import **NXPhoneCall.h**:

```
#import <phonekit/NXPhoneCall.h>
```

However, for most of NeXTSTEP API, there's a simpler and more efficient path. Some of the software kits have a master header file that imports all the other header files required by that kit. Matched to the master header file is a parsed and precompiled version of all the header files it directly or indirectly includes. By importing the master file, you get the header files in their precompiled form. This saves the compiler several steps, and a great deal of time. It's much more efficient than importing individual header files for each part of the API you use.

The following table lists the master files that correspond to precompiled versions of the header files.

Header File	Contents
appkit/appkit.h	Application Kit, Sound Kit™, all the common classes, and most of the common functions
3Dkit/3Dkit.h	3D Graphics Kit
dbkit/dbkit.h	Database Kit

All three of these files also include the root Object class (through the normal process of Objective C inheritance).

How the Manual Is Organized

Each chapter of the *General Reference* is devoted to a separate software kit or a separate group of functionally related classes and functions. The chapters are:

- Chapter 1, "Root Class," describes the Object class, the class that stands at the root of almost all Objective C inheritance hierarchies. It's the one class that all other classes inherit from and the class that all NeXTSTEP software kits are based upon.
- Chapter 2, "Application Kit," describes the basic software for writing interactive applications—applications that use windows, draw on the screen, and respond to user actions on the keyboard and mouse. The Application Kit contains the fundamental building blocks for the NeXTSTEP user interface.
- Chapter 3, "Common Classes and Functions," describes an assortment of classes and functions that aid applications in managing data and resources. These facilities are used by a wide variety of applications and range from storage allocators and hashing routines to error handling and language localization aids.
- Chapter 4, "Database Kit," describes a software kit that enables applications to communicate with database servers, such as those provided by Oracle or Sybase, using a high-level entity-relationship model. The kit provides record management, buffering, and modeling services, as well as user-interface objects for displaying and editing data.

- Chapter 5, "Display PostScript®," describes the NeXTSTEP implementation of the Display PostScript Client Library. The Client Library is mainly documented by Adobe Systems, Inc. (see "Suggested Reading" at the end of Volume 2), but NeXTSTEP has extended the Library in various ways. This chapter documents those extensions.

- Chapter 6, "Distributed Objects," describes how Objective C messages can be sent between remote objects—objects in different tasks or in different threads of the same task. A distributed objects architecture makes it possible to have different applications cooperate on a single project at run time or to split an application into various independent processes.

- Chapter 7, "Indexing Kit™," describes a set of tools for manipulating large or small amounts of data—especially for retrieving data items, based on their contents, from a large store. Among other things, the Indexing Kit can be used to build flat-file databases or to create applications (like Digital Librarian™) that search for text in collections of files or database records.

- Chapter 8, "Interface Builder™," describes the programming interface to Interface Builder, the application that enables you to design an application graphically on-screen. The chapter shows how to use this API to augment Interface Builder's standard set of tools. You can create loadable palettes containing your own custom objects and provide custom inspectors and editors for these objects. With this API, you'll be able to adapt Interface Builder to any number of highly specific uses. For a tutorial on creating a simple loadable palette and inspector, see the *NeXTSTEP Development Tools and Techniques* manual.

- Chapter 9, "Mach Kit™," describes an Objective C interface to a part of the Mach operating system. A portion of this interface is used by the distributed objects architecture documented in Chapter 6.

- Chapter 10, "MIDI," describes the functions that control the MIDI (Musical Instrument Digital Interface) device driver. The device driver manages the flow of MIDI data to and from an external device, such as a synthesizer, digital piano, or another computer.

- Chapter 11, "NetInfo Kit™," describes a software kit that's used to build network management applications.

- Chapter 12, "Networks: Novell® NetWare®," contains information on using Novell NetWare to connect NeXTSTEP machines.

- Chapter 13, "Phone Kit™," describes how to hook up your application to a telephone line, to make and answer calls, and to transmit and receive information during a call. When the phone line is an ISDN (Integrated Services Digital Network) line, data can be transmitted and received, without using a modem, at 64 kilobits per second per channel.

- Chapter 14, "Preferences," describes the programming interface to the Preferences application. With this interface, you can add new display modules to the application and thus extend the choices that Preferences presents to the user.

- Chapter 15, "Run-Time System," describes the run-time system for the Objective C language. For the most part, you don't have to be concerned with the API documented in this chapter unless you're developing interfaces to the run-time system other than Objective C. However, some run-time functions may be generally useful within Objective C programs.

- Chapter 16, "Sound," describes the Sound Kit and sound functions that permit applications to record, play, display, and manipulate sounds. It also includes the API to the sound driver.

- Chapter 17, "3D Graphics Kit," describes an Objective C interface for using Interactive RenderMan™. The 3D Graphics Kit works within the drawing context provided by the Application Kit, but sets up its own compatible context for rendering, manipulating, and allowing users to manipulate three-dimensional images.

- Chapter 18, "Video Class," describes the NXLiveVideoView class. An NXLiveVideoView can display live video images on-screen and record images for video display.

- Chapter 19, "Workspace Manager™," describes how you can augment Workspace Manager's standard set of contents inspectors with those of your own creation. For example, Workspace Manager comes with inspectors that show the contents of files in Rich Text Format® (RTF) and Tag Image File Format (TIFF), but doesn't necessarily provide inspectors for the data formats you'll be using in the application you write. Using the API and techniques described in this chapter, you can create content inspectors for those formats.

- Appendix A, "Data Formats," describes the standard data formats supported by NeXTSTEP. These formats permit different applications to exchange data through the pasteboard.

- Appendix B, "Default Parameters," lists the standard default parameters that affect NeXTSTEP software. Most default parameters record user preferences—for example, what font to use in menus. Some make hidden behavior visible—for example, by recording all PostScript output to the Window Server—and are therefore useful during debugging. Default parameters are read and written using functions documented in Chapter 3.
- Appendix C, "Keyboard Event information," describes the keyboard codes for NeXTSTEP encoding.
- Appendix D, "System Bitmaps," shows the bitmap images that are available with the system.
- Appendix E, "Details of the DSP," lists technical information about the DSP (digital signal processor).

How the Chapters Are Organized

Each chapter begins by listing three pieces of information of chapter-wide significance:

Library: The library that contains all the software described in the chapter. An "_s" at the end of the library name indicates that the library is shared. Code from a shared library isn't incorporated into your program. Instead, the library is mapped into the address space of your application when your application runs.

Header File Directory: The directory or directories where the API described in the chapter is declared.

Import: The header file that, directly or indirectly, includes all the header files required for using the kit. By importing this one header file, you get precompiled versions of all the header files it includes. This dramatically reduces the time required to compile an application. See "Precompiled Header Files" above.

After these three headings, the chapter is divided into a few standard sections:

Introduction

The introduction gives a broad overview of the software documented in the chapter. It describes the facilities available in the kit and how the various pieces fit together. It may also contain information about how to use particular methods and functions.

Classes

This section contains a full specification for each class defined in the kit. Classes are presented alphabetically. The structure of a class specification is described under "Classes" below.

Protocols

This section describes both formal protocols (those declared using the **@protocol** directive) and informal ones (those declared as categories). Protocol specifications resemble class specifications and are described under "Protocols" below.

Functions

Functions (and macros resembling functions) are documented next. The format for function descriptions is explained under "Functions" below.

Types and Constants

This section describes the defined types, symbolic constants, enumerations, structures, unions, and global variables that are provided as part of the kit. This API supports the classes and functions defined in the kit. See "Types and Constants" below for a description of the formats.

Other Features

If a kit has features that are not fully documented in the preceding sections, this section has notes explaining them. For example, the Application Kit chapter includes notes on how to advertise a service.

Classes

Information about a class is presented under the following headings shown in bold. The text accompanying each bold item describes the content of that particular section of the class specification.

Inherits From:

The inheritance hierarchy for the class. For example:

Panel : Window : Responder : Object

The first class listed (Panel, in this example) is the class's superclass. The last class listed is always Object, the root of all NeXTSTEP inheritance hierarchies. The classes between show the chain of inheritance from Object to the superclass. (This particular example shows the inheritance hierarchy for the Menu class of the Application Kit.)

Conforms To:

The formal protocols that the class conforms to. These include both protocols the class adopts and those it inherits from other adopting classes. If inherited, the name of the adopting class in given in parentheses. For example:

IXPostingExchange
IXPostingOperations
IXCursorPositioning (IXBTreeCursor)

(This particular example is from the IXPostingCursor class, a subclass of IXBTreeCursor in the Indexing Kit.)

Declared In:

The header file that declares the class interface. For example:

video/NXLiveVideoView.h

(This example is from the NXLiveVideoView class, which is declared in the **video** subdirectory of **/NextDeveloper/Headers**.)

Class Description

This section gives a general description of the class. It explains how the class fits into the overall design of the kit and how your application can make use of it. A class description often has information relevant to the way particular methods should be used.

Instance Variables

This section shows the instance variables declared for the class (exclusive of any private instance variables). For example, here are the instance variables declared in the List common class:

```
id *dataPtr;
unsigned int numElements;
unsigned int maxElements;
```

It then gives a short explanation for each variable.

dataPtr	The data managed by the List object (the array of objects).
numElements	The actual number of objects in the array.
maxElements	The total number of objects that can fit in currently allocated memory.

Instance variables that are for the internal use of the class are neither listed nor explained. These internal variables all begin with an underscore ("_") to prevent collisions with names that you might choose for instance variables in a subclass you define, or they are marked **@private** in the interface file.

Adopted Protocols

If the class adopts any protocols, the names of the methods declared in the protocols are listed next. These methods are normally not documented elsewhere in the class specification. Refer to the protocol specification for a complete description of these methods, their arguments, and their return types.

Method Types

Next, the methods the class declares and implements are listed by name and grouped by type. For example, methods used to draw are listed separately from methods used to handle events. This directory includes all the principal methods defined in the class (except those declared in adopted protocols) and a few that are inherited from other classes. Inherited methods are followed by the name of the class where they're defined; they're included in the directory to let you know which inherited methods you might commonly use with instances of the class and where to look for a description of those methods.

Class Methods
Instance Methods

A detailed description of each method defined in the class follows the classification by type. Methods that are used by class objects are presented first followed by methods that are used by instances. The descriptions within each group are ordered alphabetically by method name.

Each description begins with the syntax of the method's arguments and return values, continues with an explanation of the method, and ends, where appropriate, with a list of other related methods. Where a related method is defined in another class, it's followed by the name of the other class within parentheses. For example, here's a method description from the Window class:

gState

– (int)**gState**

Returns the PostScript graphics state object associated with the Window.

See also: – **gState** (View)

Internal methods used to implement the class aren't listed in the specification. Since you shouldn't override any of these methods, or use them in a message, they're excluded from both the method directory and the method descriptions. However, you may encounter them when looking at the call stack of your program from within the debugger. A private method is easily recognizable by the underscore ("_") that begins its name.

Methods Implemented by the Delegate

If a class lets you define another object—a delegate—that can intercede on behalf of instances of the class, the methods that the delegate can implement are described in a separate section. These are not methods defined in the class; rather, they're methods that you can define to respond to messages sent from instances of the class.

If you define one of these methods, the delegate will receive automatic messages to perform it at the appropriate time. For example, if you define a **windowDidBecomeKey:** method for a Window's delegate, the delegate will receive **windowDidBecomeKey:** messages whenever the Window becomes the key window. Messages are sent only if you define a method that can respond.

In essence, this section documents an informal protocol. But because these methods are so closely tied to the behavior of a particular class, they're documented with the class rather than in the "Protocols" section.

Some class specifications have separate sections with titles such as "Methods Implemented by the Superview" or "Methods Implemented by the Owner." These are also informal protocols. They document methods that can or must be implemented to receive messages on behalf of instances of the class.

Protocols

The protocols section documents both formal and informal protocols. Formal protocols are those that are declared using the **@protocol** compiler directive. They can be formally adopted and implemented by a class and tested by sending an object a **conformsTo:** message.

Some formal protocols are adopted and implemented by classes in the NeXTSTEP software kits. However, many formal protocols are declared by a kit, but not implemented by it. They list methods that you can implement to respond to kit-generated messages.

A few formal protocols are implemented by a kit, but not by a class that's part of the documented API. Rather, the protocol is implemented by an anonymous object that the kit supplies. The protocol lets you know what messages you can send to the object.

Like formal protocols, informal protocols declare a list of methods that others are invited to implement. If an informal protocol is closely associated with one particular class—for example, the list of methods implemented by the delegate—it's documented in the class description. Informal protocols associated with more than one class, or not associated with any particular class, are documented with the formal protocols in this section.

Protocol information is organized into many of the same sections as described above for a class specification. But protocols are not classes and therefore differ somewhat in the kind of information provided. The sections of a protocol specification are shown in bold below:

Adopted By: A list of the NeXTSTEP classes that adopt the protocol. Many protocols declare methods that applications must implement and so are not adopted by any NeXTSTEP classes.

Some protocols are implemented by anonymous objects (instances of an unknown class); the protocol is the only information available about what messages the object can respond to. Protocols that have an implementation available through an anonymous object generally don't have to be reimplemented by other classes.

Incorporates: Other protocols that the protocol being described incorporates by reference. One protocol incorporates others by listing them within angle brackets:

```
@protocol biathlon <skiing, shooting>
```

The protocol specification doesn't describe methods declared in incorporated protocols. See the specification for the incorporated protocol for a description of its methods.

An informal protocol can't be formally adopted by a class and it can't formally incorporate another protocol. So its description begins with information about the category where it's declared:

Category Of: The class that the category belongs to. Informal protocols are typically declared as categories of the Object class. This gives them the widest possible scope.

Both formal and informal protocols include a cross reference to a header file in **/NextDeveloper/Headers**:

Declared In: The header file where the protocol is declared.

Following this introductory information, the protocol specification is divided into only a small number of sections:

Protocol Description
Category Description

First, there's a short description of the protocol (or the category of an informal protocol). This description includes information on the purpose of the protocol and whether or not you might need to implement it.

Method Types

If the protocol includes enough methods to warrant it, they're divided by type and presented just as the methods of a class are.

Class Methods
Instance Methods

The main part of a protocol specification is the description of the methods it declares. Since these methods aren't necessarily implemented, the descriptions focus on the intent of the method. If the protocol is adopted by any NeXTSTEP classes, there may also be notes on how particular classes implement the methods.

Functions

Related functions are grouped together and the groups are arranged alphabetically by the name of the first function in each. There are cross references so that you can look up any function and find the group where it's documented.

The description of each function group is divided into a number of standard sections:

SUMMARY A brief description of the purpose of the functions.

DECLARED IN The header file where the functions are declared. If the header file is included in a master header file that has been precompiled, it's always more efficient to import the master file than to directly import the header file that declares the functions. If there is a master header file, it's listed at the beginning of the chapter under "Import."

SYNOPSIS A prototype of the functions, showing their names, return types, argument types, and calling sequence.

DESCRIPTION A description of the functions and how to use them.

If relevant, the following sections may also be present:

EXAMPLES Example code showing how the functions are used.

RETURN A statement or restatement of what each function returns.

EXCEPTIONS The exceptions that the functions might potentially raise.

SEE ALSO References to other functions or to other parts of the NeXTSTEP API.

Types and Constants

The "Types and Constants" section is divided into the following parts:

Defined Types

Types that are defined with the **typedef** compiler directive.

Symbolic Constants

Constants that are defined with the **#define** preprocessor directive. Function-like macros are documented in the "Functions" section.

Enumerations

Constants that are defined with **enum**, excluding those that are members of a defined type.

Structures

Structures and unions, excluding those that are defined types.

Globals

Global variables.

Within these subsections, each element of the API is presented in a way reminiscent of the function descriptions:

DECLARED IN The header file where the type, constant, structure, or global variable is declared.

SYNOPSIS The declaration of the type, enumerated constants, structure, or global variable, or a table listing symbolic constants. Private fields of a structure—those that begin with an underscore—are not shown. Fields of a structure that are shown but are not in bold should not be accessed or modified in application code.

DESCRIPTION A description of each part of the public API.

SEE ALSO References to other parts of the API

Conventions

Where this manual discusses functions, methods, or other programming elements, it makes special use of ellipsis, square brackets [ ], and bold and italic fonts.

Bold denotes words or characters that are to be taken literally (typed as they appear). *Italic* denotes words that represent something else or can be varied. For example, the syntax

print *expression*

means that you follow the word **print** with any expression.

Square brackets [ ] mean that the enclosed elements are optional, except when the brackets are bold **[ ]**, in which case they're to be taken literally. The exceptions are few and will be clear from the context. For example,

pointer [*filename*]

means that you specify a pointer with or without a file name after it, but

[*receiver message***]**

means that you specify a receiver and a message enclosed in square brackets.

Ellipsis (...) indicates that the previous element may be repeated. For example:

Syntax	Allows
pointer ...	One or more pointers
pointer [, *pointer*] ...	One or more pointers separated by commas
pointer [*filename* ...]	A pointer optionally followed by one or more file names
pointer [, *filename*] ...	A pointer optionally followed by a comma and one or more file names separated by commas

1 *Root Class*

1 *Root Class*

Library:	libsys_s.a
Header File Directory:	/NextDeveloper/Headers/objc
Import:	objc/Object.h, appkit/appkit.h, dbkit/dbkit.h, 3Dkit/3Dkit.h, or the interface file of any class that inherits from Object

Introduction

In the Objective C language, new classes are created as subclasses of an existing class:

```
@interface NewClass : OldClass
```

But not every class can be a subclass. The inheritance hierarchy has to start somewhere. There has to be at least one *root class*, a class that doesn't inherit from any other class:

```
@interface RootClass
```

Theoretically, there can be many different root classes, a separate one for each project or kit perhaps, or one for each group of closely related classes. However, in practice, all Objective C inheritance hierarchies are rooted in the same class—the Object class. As you look at the inheritance diagrams for the various software kits documented in this book, you'll notice that each one begins with the Object class. For example, the figure on the next page shows the Object class and part of the Application Kit inheritance hierarchy.

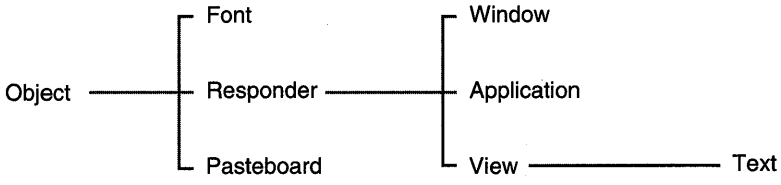


Figure 1-1. Some Application Kit Classes

Because all classes inherit from the Object class, it can define only general properties that all objects share. These shared properties are the ones that connect objects to the run-time system and enable them to behave as objects. For example, the Object class gives all objects the ability to identify their class and to find which method to use in response to a message. It provides class objects with methods to create new instances, and instances with methods to forward messages and archive and copy themselves. In short, the Object class defines what it is to be an Objective C object.

It's precisely for this reason that Object is used as the universal root class. There's no point in reinventing object-oriented behavior each time you develop a new class. It's better to declare a new class as a subclass of Object, or of another class that inherits from Object.

The Object class is the root class used by all NeXTSTEP software kits and the one that should be used in all NeXTSTEP applications. However, NeXTSTEP includes one other root class for a special purpose. The NXProxy class, described in Chapter 6, "Distributed Objects," defines an object that can stand in for, and assume the identity of, another object, one located in a remote process. By sending messages to the proxy, an application can in fact communicate with the remote object. NXProxy is a root class only because proxy objects need to behave differently from all other objects; they can't inherit typical object behavior. Except for special cases like this, all ordinary objects should inherit from the Object class.

Classes

Object

Inherits From: none *(Object is the root class)*

Declared In: objc/Object.h

Class Description

Object is the root class of all ordinary Objective C inheritance hierarchies; it's the one class that has no superclass. From Object, other classes inherit a basic interface to the run-time system for the Objective C language. It's through Object that instances of all classes obtain their ability to behave as objects.

Among other things, the Object class provides inheriting classes with a framework for creating, initializing, freeing, copying, comparing, and archiving objects, for performing methods selected at run-time, for querying an object about its methods and its position in the inheritance hierarchy, and for forwarding messages to other objects. For example, to ask an object what class it belongs to, you'd send it a **class** message. To find out whether it implements a particular method, you'd send it a **respondsTo:** message.

The Object class is an abstract class; programs use instances of classes that inherit from Object, but never of Object itself.

Initializing an Object to Its Class

Every object is connected to the run-time system through its **isa** instance variable, inherited from the Object class. **isa** identifies the object's class; it points to a structure that's compiled from the class definition. Through **isa**, an object can find whatever information it needs at run time—such as its place in the inheritance hierarchy, the size and structure of its instance variables, and the location of the method implementations it can perform in response to messages.

Because all objects directly or indirectly inherit from the Object class, they all have this variable. The defining characteristic of an "object" is that its first instance variable is an **isa** pointer to a class structure.

The installation of the class structure—the initialization of **isa**—is one of the responsibilities of the **alloc**, **allocFromZone:**, and **new** methods, the same methods that

create (allocate memory for) new instances of a class. In other words, class initialization is part of the process of creating an object; it's not left to the methods, such as **init**, that initialize individual objects with their particular characteristics.

Instance and Class Methods

Every object requires an interface to the run-time system, whether it's an instance object or a class object. For example, it should be possible to ask either an instance or a class about its position in the inheritance hierarchy or whether it can respond to a particular message.

So that this won't mean implementing every Object method twice, once as an instance method and again as a class method, the run-time system treats methods defined in the root class in a special way:

> *Instance methods defined in the root class can be performed both by instances and by class objects.*

A class object has access to class methods—those defined in the class and those inheritied from the classes above it in the inheritance hierarchy—but generally not to instance methods. However, the run-time system gives all class objects access to the instance methods defined in the root class. Any class object can perform any root instance method, provided it doesn't have a class method with the same name.

For example, a class object could be sent messages to perform Object's **respondsTo:** and **perform:with:** instance methods:

```
SEL method = @selector(riskAll:);

if ( [MyClass respondsTo:method] )
    [MyClass perform:method with:self];
```

When a class object receives a message, the run-time system looks first at the receiver's repertoire of class methods. If it fails to find a class method that can respond to the message, it looks at the set of instance methods defined in the root class. If the root class has an instance method that can respond (as Object does for **respondsTo:** and **perform:with:**), the run-time system uses that implementation and the message succeeds.

Note that the only instance methods available to a class object are those defined in the root class. If MyClass in the example above had reimplemented either **respondsTo:** or **perform:with:**, those new versions of the methods would be available only to instances. The class object for MyClass could perform only the versions defined in the Object class. (Of course, if MyClass had implemented **respondsTo:** or **perform:with:** as class methods rather than instance methods, the class would perform those new versions.)

Interface Conventions

The Object class defines a number of methods that other classes are expected to override. Often, Object's default implementation simply returns **self**. Putting these "empty" methods in the Object class serves two purposes:

- It means that every object can readily respond to certain standard messages, such as **awake** or **init**, even if the response is to do nothing It's not necessary to check (using **respondsTo:**) before sending the message.
- It establishes conventions that, when followed by all classes, make object interactions more reliable. These conventions are explained in full under the method descriptions.

Sometimes a method is merely declared in the Object class; it has no implementation, not even the empty one of returning **self**. These "unimplemented" methods serve the same purpose—defining an interface convention—as Object's "empty" methods. When implemented, they enable objects to respond to system-generated messages.

Instance Variables

Class **isa**;

isa	A pointer to the instance's class structure.

Method Types

Initializing the class	+ initialize
Creating, copying, and freeing instances	+ alloc + allocFromZone: + new – copy – copyFromZone: – zone – free + free
Initializing a new instance	– init

Identifying classes	+ name
	+ class
	– class
	+ superclass
	– superclass
Identifying and comparing instances	
	– isEqual:
	– hash
	– self
	– name
	– printForDebugger:
Testing inheritance relationships	
	– isKindOf:
	– isKindOfClassNamed:
	– isMemberOf:
	– isMemberOfClassNamed:
Testing class functionality	– respondsTo:
	+ instancesRespondTo:
Testing for protocol conformance	
	+ conformsTo:
	– conformsTo:
Sending messages determined at run time	
	– perform:
	– perform:with:
	– perform:with:with:
Forwarding messages	– forward::
	– performv::
Obtaining method information	– methodFor:
	+ instanceMethodFor:
	– descriptionForMethod:
	+ descriptionForInstanceMethod:
Posing	+ poseAs:
Enforcing intentions	– notImplemented:
	– subclassResponsibility:
Error handling	– doesNotRecognize:
	– error:
Dynamic loading	+ finishLoading:
	+ startUnloading

Archiving	– read: – write: – startArchiving: – awake – finishUnarchiving + setVersion: + version

Class Methods

alloc

+ alloc

Returns a new instance of the receiving class. The **isa** instance variable of the new object is initialized to a data structure that describes the class; memory for all other instance variables is set to 0. A version of the **init** method should be used to complete the initialization process. For example:

```
id newObject = [[TheClass alloc] init];
```

Other classes shouldn't override **alloc** to add code that initializes the new instance. Instead, class-specific versions of the **init** method should be implemented for that purpose. Versions of the **new** method can also be implemented to combine allocation and initialization.

Note: The **alloc** method doesn't invoke **allocFromZone:**. The two methods work independently.

See also: + allocFromZone:, – init, + new

allocFromZone:

+ allocFromZone:(NXZone *)*zone*

Returns a new instance of the receiving class. Memory for the new object is allocated from *zone*.

The **isa** instance variable of the new object is initialized to a data structure that describes the class; memory for its other instance variables is set to 0. A version of the **init** method should be used to complete the initialization process. For example:

```
id newObject = [[TheClass allocFromZone:someZone] init];
```

The **allocFromZone:** method shouldn't be overridden to include any initialization code. Instead, class-specific versions of the **init** method should be implemented for that purpose.

When one object creates another, it's often a good idea to make sure they're both allocated from the same region of memory. The **zone** method can be used for this purpose; it returns the zone where the receiver is located. For example:

```
id myCompanion = [[TheClass allocFromZone:[self zone]] init];
```

See also: + alloc, – zone, – init

class

+ class

Returns **self**. Since this is a class method, it returns the class object.

When a class is the receiver of a message, it can be referred to by name. In all other cases, the class object must be obtained through this, or a similar method. For example, here SomeClass is passed as an argument to the **isKindOf:** method:

```
BOOL test = [self isKindOf:[SomeClass class]];
```

See also: – name, – class

conformsTo:

+ (BOOL)**conformsTo:**(Protocol *)*aProtocol*

Returns YES if the receiving class conforms to *aProtocol*, and NO if it doesn't.

A class is said to "conform to" a protocol if it adopts the protocol or inherits from another class that adopts it. Protocols are adopted by listing them within angle brackets after the interface declaration. Here, for example, MyClass adopts the imaginary AffiliationRequests and Normalization protocols:

```
@interface MyClass : Object <AffiliationRequests, Normalization>
```

A class also conforms to any protocols that are incorporated in the protocols it adopts or inherits. Protocols incorporate other protocols in the same way that classes adopt them. For example, here the AffiliationRequests protocol incorporates the Joining protocol:

```
@protocol AffiliationRequests <Joining>
```

When a class adopts a protocol, it must implement all the methods declared in the protocol (and in any protocols incorporated in the protocol it adopts). In the example above, MyClass must implement the methods in the AffiliationRequests, Joining, and Normalization protocols. When this convention is followed and all the methods in adopted protocols are in fact implemented, the **conformsTo:** test for a set of methods becomes roughly equivalent to the **respondsTo:** test for a single method.

However, this method judges conformance solely on the basis of the formal declarations in source code, as illustrated above. It doesn't check to see whether the methods declared in the protocol are actually implemented. It's the programmer's responsibility to see that they are.

The Protocol object required as this method's argument can be specified using the **@protocol()** directive:

```
BOOL canJoin = [MyClass conformsTo:@protocol(Joining)]
```

The Protocol class is documented in Chapter 15, "Run-Time System."

See also: – conformsTo:

descriptionForInstanceMethod:

+ (struct objc_method_description *)
    descriptionForInstanceMethod:(SEL)*aSelector*

Returns a pointer to a structure that describes the *aSelector* instance method, or NULL if the *aSelector* method can't be found. To ask the class for a description of a class method, or an instance for the description of an instance method, use the **descriptionForMethod:** instance method.

See also: – descriptionForMethod:

finishLoading:

+ **finishLoading:**(struct mach_header *)*header*

Implemented by subclasses to integrate the class, or a category of the class, into a running program. A **finishLoading:** message is sent immediately after the class or category has been dynamically loaded into memory, but only if the newly loaded class or category implements a method that can respond. *header* is a pointer to the structure that describes the modules that were just loaded.

Once a dynamically loaded class is used, it will also receive an **initialize** message. However, because the **finishLoading:** message is sent immediately after the class is loaded, it always precedes the **initialize** message, which is sent only when the class receives its first message from within the program.

A **finishLoading:** method is specific to the class or category where it's defined; it's not inherited by subclasses or shared with the rest of the class. Thus a class that has four categories can define a total of five **finishLoading:** methods, one in each category and one in the main class definition. The method that's performed is the one defined in the class or category just loaded.

There's no default **finishLoading:** method. The Object class declares a prototype for this method, but doesn't implement it.

See also: + startUnloading

free

+ free

Returns **nil**. This method is implemented to prevent class objects, which are "owned" by the run-time system, from being accidentally freed. To free an instance, use the instance method **free**.

See also: – free

initialize

+ initialize

Initializes the class before it's used (before it receives its first message). The run-time system generates an **initialize** message to each class just before the class, or any class that inherits from it, is sent its first message from within the program. Each class object receives the **initialize** message just once. Superclasses receive it before subclasses do.

For example, if the first message your program sends is this,

```
[Application new]
```

the run-time system will generate these three **initialize** messages,

```
[Object initialize];
[Responder initialize];
[Application initialize];
```

since Application is a subclass of Responder and Responder is a subclass of Object. All the **initialize** messages precede the **new** message and are sent in the order of inheritance, as shown.

If your program later begins to use the Text class,

```
[Text instancesRespondTo:someSelector]
```

the run-time system will generate these additional **initialize** messages,

```
[View initialize];
[Text initialize];
```

since the Text class inherits from Object, Responder, and View. The **instancesRespondTo:** message is sent only after all these classes are initialized. Note that the **initialize** messages to Object and Responder aren't repeated; each class is initialized only once.

You can implement your own versions of **initialize** to provide class-specific initialization as needed.

Because **initialize** methods are inherited, it's possible for the same method to be invoked many times, once for the class that defines it and once for each inheriting class. To prevent code from being repeated each time the method is invoked, it can be bracketed as shown in the example below:

```
+ initialize
{
    if ( self == [MyClass class] ) {
        /* put initialization code here */
    }
    return self;
}
```

Since the run-time system sends a class just one **initialize** message, the test shown in the example above should prevent code from being invoked more than once. However, if for some reason an application also generates **initialize** messages, a more explicit test may be needed:

```
+ initialize
{
    static BOOL tooLate = NO;
    if ( !tooLate ) {
        /* put initialization code here */
        tooLate = YES;
    }
    return self;
}
```

See also: – init, – class

instanceMethodFor:

+ (IMP)**instanceMethodFor:**(SEL)*aSelector*

Locates and returns the address of the implementation of the *aSelector* instance method. An error is generated if instances of the receiver can't respond to *aSelector* messages.

This method is used to ask the class object for the implementation of an instance method. To ask the class for the implementation of a class method, use the instance method **methodFor:** instead of this one.

instanceMethodFor:, and the function pointer it returns, are subject to the same constraints as those described for **methodFor:**.

See also: – **methodFor:**

instancesRespondTo:

+ (BOOL)**instancesRespondTo:**(SEL)*aSelector*

Returns YES if instances of the class are capable of responding to *aSelector* messages, and NO if they're not. To ask the class whether it, rather than its instances, can respond to a particular message, use the **respondsTo:** instance method instead of **instancesRespondTo:**.

If *aSelector* messages are forwarded to other objects, instances of the class will be able to receive those messages without error even though this method returns NO.

See also: – **respondsTo:**, – **forward::**

name

+ (const char *)**name**

Returns a null-terminated string containing the name of the class. This information is often used in error messages or debugging statements.

See also: – **name**, + **class**

new

+ **new**

Creates a new instance of the receiving class, sends it an **init** message, and returns the initialized object returned by **init**.

As defined in the Object class, **new** is essentially a combination of **alloc** and **init**. Like **alloc**, it initializes the **isa** instance variable of the new object so that it points to the class data structure. It then invokes the **init** method to complete the initialization process.

Unlike **alloc**, **new** is sometimes reimplemented in subclasses to have it invoke a class-specific initialization method. If the **init** method includes arguments, they're typically reflected in the **new** method as well. For example:

```
+ newArg:(int)tag arg:(struct info *)data
{
    return [[self alloc] initArg:tag arg:data];
}
```

However, there's little point in implementing a **new...** method if it's simply a shorthand for **alloc** and **init...**, like the one shown above. Often **new...** methods will do more than just allocation and initialization. In some classes, they manage a set of instances, returning the one with the requested properties if it already exists, allocating and initializing a new one only if necessary. For example:

```
+ newArg:(int)tag arg:(struct info *)data
{
    id theInstance;

    if ( theInstance = findTheObjectWithTheTag(tag) )
        return theInstance;
    return [[self alloc] initArg:tag arg:data];
}
```

Although it's appropriate to define new **new...** methods in this way, the **alloc** and **allocFromZone:** methods should never be augmented to include initialization code.

See also: **– init**, **+ alloc**, **+ allocFromZone:**

poseAs:

+ poseAs:*aClassObject*

Causes the receiving class to "pose as" its superclass, the *aClassObject* class. The receiver takes the place of *aClassObject* in the inheritance hierarchy; all messages sent to *aClassObject* will actually be delivered to the receiver. The receiver must be defined as a subclass of *aClassObject*. It can't declare any new instance variables of its own, but it can define new methods and override methods defined in the superclass. The **poseAs:** message should be sent before any messages are sent to *aClassObject* and before any instances of *aClassObject* are created.

This facility allows you to add methods to an existing class by defining them in a subclass and having the subclass substitute for the existing class. The new method definitions will be inherited by all subclasses of the superclass. Care should be taken to ensure that this doesn't generate errors.

A subclass that poses as its superclass still inherits from the superclass. Therefore, none of the functionality of the superclass is lost in the substitution. Posing doesn't alter the definition of either class.

Posing is useful as a debugging tool, but category definitions are a less complicated and more efficient way of augmenting existing classes. Posing admits only two possibilities that are absent for categories:

- A method defined by a posing class can override any method defined by its superclass. Methods defined in categories can replace methods defined in the class proper, but they cannot reliably replace methods defined in other categories. If two categories define the same method, one of the definitions will prevail, but there's no guarantee which one.
- A method defined by a posing class can, through a message to **super**, incorporate the superclass method it overrides. A method defined in a category can replace a method defined elsewhere by the class, but it can't incorporate the method it replaces.

If successful, this method returns **self**. If not, it generates an error message and aborts.

setVersion:

+ **setVersion:**(int)*aVersion*

Sets the class version number to *aVersion*, and returns **self**. The version number is helpful when instances of the class are to be archived and reused later. The default version is 0.

See also: + **version**

startUnloading

+ **startUnloading**

Implemented by subclasses to prepare for the class, or a category of the class, being unloaded from a running program. A **startUnloading** message is sent immediately before the class or category is unloaded, but only if the class or category about to be unloaded implements a method that can respond.

A **startUnloading** method is specific to the class or category where it's defined; it isn't inherited by subclasses or shared with the rest of the class. Thus a class that has four categories can define a total of five **startUnloading** methods, one in each category and one in the main class definition. The method that's performed is the one defined in the class or category that will be unloaded.

There's no default **startUnloading** method. The object class declares a prototype for this method but doesn't implement it.

See also: + **finishLoading:**

superclass

+ **superclass**

Returns the class object for the receiver's superclass.

See also: + **class**, – **superclass**

version

+ (int)**version**

Returns the version number assigned to the class. If no version has been set, this will be 0.

See also: + **setVersion:**

Instance Methods

awake

– **awake**

Implemented by subclasses to reinitialize the receiving object after it has been unarchived (by **read:**). An **awake** message is automatically sent to every object after it has been unarchived and after all the objects it refers to are in a usable state.

The default version of the method defined here merely returns **self**.

A class can implement an **awake** method to provide for more initialization than can be done in the **read:** method. Each implementation of **awake** should limit the work it does to the scope of the class definition, and incorporate the initialization of classes farther up the inheritance hierarchy through a message to **super**. For example:

```
- awake
{
    [super awake];
    /* class-specific initialization goes here */
    return self;
}
```

All implementations of **awake** should return **self**.

Note: Not all objects loaded from a nib file (created by Interface Builder) are unarchived; some are newly instantiated. Those that are unarchived receive an **awake** message, but those that are instantiated do not. See the Interface Builder documentation in *NeXTSTEP Development Tools* for more information.

See also: – **read:**, – **finishUnarchiving**, – **awakeFromNib** (NXNibNotification protocol in the Application Kit), – **loadNibFile:owner:** (Application class in the Application Kit)

class

– **class**

Returns the class object for the receiver's class.

See also: + **class**

conformsTo:

– (BOOL)**conformsTo:**(Protocol *)*aProtocol*

Returns YES if the class of the receiver conforms to *aProtocol*, and NO if it doesn't. This method invokes the **conformsTo:** class method to do its work. It's provided as a convenience so that you don't need to get the class object to find out whether an instance can respond to a given set of messages.

See also: + **conformsTo:**

copy

– **copy**

Returns a new instance that's an exact copy of the receiver. This method creates only one new object. If the receiver has instance variables that point to other objects, the instance variables in the copy will point to the same objects. The values of the instance variables are copied, but the objects they point to are not.

This method does its work by invoking the **copyFromZone:** method and specifying that the copy should be allocated from the same memory zone as the receiver. If a subclass implements its own **copyFromZone:** method, this **copy** method will use it to copy instances of the subclass. Therefore, a class can support copying from both methods just by implementing a class-specific version of **copyFromZone:**.

See also: – **copyFromZone:**

copyFromZone:

– **copyFromZone:**(NXZone *)*zone*

Returns a new instance that's an exact copy of the receiver. Memory for the new instance is allocated from *zone*.

This method creates only one new object. If the receiver has instance variables that point to other objects, the instance variables in the copy will point to the same objects. The values of the instance variables are copied, but the objects they point to are not.

Subclasses should implement their own versions of **copyFromZone:**, not **copy**, to define class-specific copying.

See also: – **copy**, – **zone**

descriptionForMethod:

– (struct objc_method_description *)**descriptionForMethod:**(SEL)*aSelector*

Returns a pointer to a structure that describes the *aSelector* method, or NULL if the *aSelector* method can't be found. When the receiver is an instance, *aSelector* should be an instance method; when the receiver is a class, it should be a class method.

The **objc_method_description** structure is declared in **objc/Protocol.h**, and is mostly used in the implementation of protocols. It includes two fields—the selector for the method (which will be the same as *aSelector*) and a character string encoding the method's return and argument types:

```
struct objc_method_description {
    SEL name;
    char *types;
};
```

Type information is encoded according to the conventions of the **@encode()** directive, but the string also includes information about total argument size and individual argument offsets. For example, if **descriptionForMethod:** were asked for a description of itself, it would return this string in the **types** field:

```
^{objc_method_description=:*}12@8:12:16
```

This records the fact that **descriptionForMethod:** returns a pointer ('^') to a structure ("{...}") and that it pushes a total of 12 bytes on the stack. The structure is called "objc_method_description" and it consists of a selector (':') and a character pointer ('*'). The first argument, **self**, is an object ('@') at an offset of 8 bytes from the stack pointer, the second argument, **_cmd**, is a selector (':') at an offset of 12 bytes, and the third argument, *aSelector*, is also a selector but at an offset of 16 bytes. The first two arguments—**self** for

the message receiver and **_cmd** for the method selector—are passed to every method implementation but are hidden by the Objective C language.

The type codes used for methods declared in a class or category are:

Meaning	Code
id	'@'
Class	'#'
SEL	':'
void	'v'
char	'c'
unsigned char	'C'
short	's'
unsigned short	'S'
int	'i'
unsigned int	'I'
long	'l'
unsigned long	'L'
float	'f'
double	'd'
char *	'*'
any other pointer	'^'
an undefined type	'?'
a bitfield	'b'
begin an array	'['
end an array	']'
begin a union	'('
end a union	')'
begin a structure	'{'
end a structure	'}'

The same codes are used for methods declared in a protocol, but with these additions for type modifiers:

const	'r'
in	'n'
inout	'N'
out	'o'
bycopy	'O'
oneway	'V'

See also: **+ descriptionForInstanceMethod:**, **– descriptionForClassMethod:** (Protocol class in the Run-Time System), **– descriptionForInstanceMethod** (Protocol class in the Run-Time System)

doesNotRecognize:

– **doesNotRecognize:**(SEL)*aSelector*

Handles *aSelector* messages that the receiver doesn't recognize. The run-time system invokes this method whenever an object receives an *aSelector* message that it can't respond to or forward. This method, in turn, invokes the **error:** method to generate an error message and abort the current process.

doesNotRecognize: messages should be sent only by the run-time system. Although they're sometimes used in program code to prevent a method from being inherited, it's better to use the **error:** method directly. For example, an Object subclass might renounce the **copy** method by reimplementing it to include an **error:** message as follows:

```
- copy
{
    [self error:"  %s objects should not be sent '%s' messages\n",
            [[self class] name], sel_getName(_cmd)];
}
```

This code prevents instances of the subclass from recognizing or forwarding **copy** messages—although the **respondsTo:** method will still report that the receiver has access to a **copy** method.

(The **_cmd** variable identifies the current selector; in the example above, it identifies the selector for the **copy** method. The **sel_getName()** function returns the method name corresponding to a selector code; in the example, it returns the name "copy".)

See also: – **error:**, – **subclassResponsibility:**, + **name**

error:

– **error:**(const char *)*aString, ...*

Generates a formatted error message, in the manner of **printf()**, from *aString* followed by a variable number of arguments. For example:

```
[self error:"index %d exceeds limit %d\n", index, limit];
```

The message specified by *aString* is preceded by this standard prefix (where *class* is the name of the receiver's class):

```
"error: class "
```

This method doesn't return. It calls the run-time **_error** function, which first generates the error message and then calls **abort()** to create a core file and terminate the process.

See also: – **subclassResponsibility:**, – **notImplemented:**, – **doesNotRecognize:**

finishUnarchiving

– **finishUnarchiving**

Implemented by subclasses to replace an unarchived object with a new object if necessary. A **finishUnarchiving** message is sent to every object after it has been unarchived (using **read:**) and initialized (by **awake**), but only if a method has been implemented that can respond to the message.

The **finishUnarchiving** message gives the application an opportunity to test an unarchived and initialized object to see whether it's usable, and, if not, to replace it with another object that is. This method should return **nil** if the unarchived instance (**self**) is OK; otherwise, it should free the receiver and return another object to take its place.

There's no default implementation of the **finishUnarchiving** method. The Object class declares this method, but doesn't define it.

See also: – **read:**, – **awake**, – **startArchiving:**

forward::

– **forward:**(SEL)*aSelector* **:**(marg_list)*argFrame*

Implemented by subclasses to forward messages to other objects. When an object is sent an *aSelector* message, and the run-time system can't find an implementation of the method for the receiving object, it sends the object a **forward::** message to give it an opportunity to delegate the message to another receiver. (If the delegated receiver can't respond to the message either, it too will be given a chance to forward it.)

The **forward::** message thus allows an object to establish relationships with other objects that will, for certain messages, act on its behalf. The forwarding object is, in a sense, able to "inherit" some of the characteristics of the object it forwards the message to.

A **forward::** message is generated only if the *aSelector* method isn't implemented by the receiving object's class or by any of the classes it inherits from.

An implementation of the **forward::** method has two tasks:

- To locate an object that can respond to the *aSelector* message. This need not be the same object for all messages.
- To send the message to that object, using the **performv::** method.

In the simple case, in which an object forwards messages to just one destination (such as the hypothetical **friend** instance variable in the example below), a **forward::** method could be as simple as this:

```
- forward:(SEL)aSelector :(marg_list)argFrame
{
    if ( [friend respondsTo:aSelector] )
        return [friend performv:aSelector :argFrame];
    [self doesNotRecognize:aSelector];
}
```

argFrame is a pointer to the arguments included in the original *aSelector* message. It's passed directly to **performv::** without change. (However, *argFrame* does not correctly capture variable arguments. Messages that include a variable argument list—for example, messages to perform Object's **error**: method—cannot be forwarded.)

(Note that in the example **forward::** returns unchanged the value returned by **performv::**. Since **forward::** returns a pointer, specifically an **id**, the *aSelector* method must also be one that returns a pointer (or **void**). Methods that return other types cannot be reliably forwarded.)

Implementations of the **forward::** method can do more than just forward messages. **forward::** can, for example, be used to consolidate code that responds to a variety of different messages, thus avoiding the necessity of having to write a separate method for each selector. A **forward::** method might also involve several other objects in the response to a given message, rather than forward it to just one.

The default version of **forward::** implemented in the Object class simply invokes the **doesNotRecognize:** method; it doesn't forward messages. Thus, if you choose not to implement **forward::**, unrecognized messages will generate an error and cause the task to abort.

See also: – performv::, – doesNotRecognize:

free

– free

Frees the memory occupied by the receiver and returns **nil**. Subsequent messages to the object will generate an error indicating that a message was sent to a freed object (provided that the freed memory hasn't been reused yet).

Subclasses must implement their own versions of **free** to deallocate any additional memory consumed by the object—such as dynamically allocated storage for data, or other objects that are tightly coupled to the freed object and are of no use without it. After performing the class-specific deallocation, the subclass method should incorporate superclass versions of **free** through a message to **super**:

```
- free {
    [companion free];
    free(privateMemory);
    vm_deallocate(task_self(), sharedMemory, memorySize);
    return [super free];
}
```

If, under special circumstances, a subclass version of **free** refuses to free the receiver, it should return **self** instead of **nil**. Object's default version of this method always frees the receiver and always returns **nil**. It calls **object_deallocate()** to accomplish the deallocation.

hash

– (unsigned int)**hash**

Returns an unsigned integer that's derived from the **id** of the receiver. The integer is guaranteed to always be the same for the same **id**.

See also: – **isEqual:**

init

– **init**

Implemented by subclasses to initialize a new object (the receiver) immediately after memory for it has been allocated. An **init** message is generally coupled with an **alloc** or **allocFromZone:** message in the same line of code:

```
id newObject = [[TheClass alloc] init];
```

An object isn't ready to be used until it has been initialized. The version of the **init** method defined in the Object class does no initialization; it simply returns **self**.

Subclass versions of this method should return the new object (**self**) after it has been successfully initialized. If it can't be initialized, they should free the object and return **nil**. In some cases, an **init** method might free the new object and return a substitute. Programs should therefore always use the object returned by **init**, and not necessarily the one returned by **alloc** or **allocFromZone:**, in subsequent code.

Every class must guarantee that the **init** method returns a fully functional instance of the class. Typically this means overriding the method to add class-specific initialization code. Subclass versions of **init** need to incorporate the initialization code for the classes they inherit from, through a message to **super**:

```
- init
{
    [super init];
    /* class-specific initialization goes here */
    return self;
}
```

Note that the message to **super** precedes the initialization code added in the method. This ensures that initialization proceeds in the order of inheritance.

Subclasses often add arguments to the **init** method to allow specific values to be set. The more arguments a method has, the more freedom it gives you to determine the character of initialized objects. Classes often have a set of **init...** methods, each with a different number of arguments. For example:

```
- init;
- initArg:(int)tag;
- initArg:(int)tag arg:(struct info *)data;
```

The convention is that at least one of these methods, usually the one with the most arguments, includes a message to **super** to incorporate the initialization of classes higher up the hierarchy. This method is the *designated initializer* for the class. The other **init...** methods defined in the class directly or indirectly invoke the designated initializer through messages to **self**. In this way, all **init...** methods are chained together. For example:

```
- init
{
    return [self initArg:-1];
}

- initArg:(int)tag
{
    return [self initArg:tag arg:NULL];
}

- initArg:(int)tag arg:(struct info *)data
{
    [super init. . .];
    /* class-specific initialization goes here */
}
```

In this example, the **initArg:arg:** method is the designated initializer for the class.

If a subclass does any initialization of its own, it must define its own designated initializer. This method should begin by sending a message to **super** to perform the designated initializer of its superclass. Suppose, for example, that the three methods illustrated above are defined in the B class. The C class, a subclass of B, might have this designated initializer:

```
- initArg:(int)tag arg:(struct info *)data arg:anObject
{
    [super initArg:tag arg:data];
    /* class-specific initialization goes here */
}
```

If inherited **init...** methods are to successfully initialize instances of the subclass, they must all be made to (directly or indirectly) invoke the new designated initializer. To accomplish this, the subclass is obliged to cover (override) only the designated initializer of the superclass. For example, in addition to its designated initializer, the C class would also implement this method:

```
- initArg:(int)tag arg:(struct info *)data
{
    return [self initArg:tag arg:data arg:nil];
}
```

This ensures that all three methods inherited from the B class also work for instances of the C class.

Often the designated initializer of the subclass overrides the designated initializer of the superclass. If so, the subclass need only implement the one **init...** method.

These conventions maintain a direct chain of **init...** links, and ensure that the **new** method and all inherited **init...** methods return usable, initialized objects. They also prevent the possibility of an infinite loop wherein a subclass method sends a message (to **super**) to perform a superclass method, which in turn sends a message (to **self**) to perform the subclass method.

This **init** method is the designated initializer for the Object class. Subclasses that do their own initialization should override it, as described above.

See also: + new, + alloc, + allocFromZone:

isEqual:

– (BOOL)**isEqual:***anObject*

Returns YES if the receiver is the same as *anObject*, and NO if it isn't. This is determined by comparing the **id** of the receiver to the **id** of *anObject*.

Subclasses may need to override this method to provide a different test of equivalence. For example, in some contexts, two objects might be said to be the same if they're both the same kind of object and they both contain the same data:

```
- (BOOL)isEqual:anObject
{
    if ( anObject == self )
        return YES;
    if ( [anObject isKindOf:[self class]] ) {
        if ( !strcmp(stringData, [anObject stringData]) )
            return YES;
    }
    return NO;
}
```

isKindOf:

– (BOOL)**isKindOf:***aClassObject*

Returns YES if the receiver is an instance of *aClassObject* or an instance of any class that inherits from *aClassObject*. Otherwise, it returns NO. For example, in this code **isKindOf:** would return YES because, in the Application Kit, the Menu class inherits from Window:

```
id  aMenu = [[Menu alloc] init];
if ( [aMenu isKindOf:[Window class]] )
    . . .
```

When the receiver is a class object, this method returns YES if *aClassObject* is the Object class, and NO otherwise.

See also: – isMemberOf:

isKindOfClassNamed:

– (BOOL)**isKindOfClassNamed:**(const char *)*aClassName*

Returns YES if the receiver is an instance of *aClassName* or an instance of any class that inherits from *aClassName*. This method is the same as **isKindOf:**, except it takes the class name, rather than the class **id**, as its argument.

See also: – **isMemberOfClassNamed:**

isMemberOf:

– (BOOL)**isMemberOf:***aClassObject*

Returns YES if the receiver is an instance of *aClassObject*. Otherwise, it returns NO. For example, in this code, **isMemberOf:** would return NO:

```
id  aMenu = [[Menu alloc] init];
if ([aMenu isMemberOf:[Window class]])
    . . .
```

When the receiver is a class object, this method returns NO. Class objects are not "members of" any class.

See also: – **isKindOf:**

isMemberOfClassNamed:

– (BOOL)**isMemberOfClassNamed:**(const char *)*aClassName*

Returns YES if the receiver is an instance of *aClassName*, and NO if it isn't. This method is the same as **isMemberOf:**, except it takes the class name, rather than the class **id**, as its argument.

See also: – **isKindOfClassNamed:**

methodFor:

– (IMP)**methodFor:**(SEL)*aSelector*

Locates and returns the address of the receiver's implementation of the *aSelector* method, so that it can be called as a function. If the receiver is an instance, *aSelector* should refer to an instance method; if the receiver is a class, it should refer to a class method.

aSelector must be a valid, nonNULL selector. If in doubt, use the **respondsTo:** method to check before passing the selector to **methodFor:**.

IMP is defined (in the **objc/objc.h** header file) as a pointer to a function that returns an **id** and takes a variable number of arguments (in addition to the two "hidden" arguments—**self** and **_cmd**—that are passed to every method implementation):

```
typedef id (*IMP)(id, SEL, ...);
```

This definition serves as a prototype for the function pointer that **methodFor:** returns. It's sufficient for methods that return an object and take object arguments. However, if the *aSelector* method takes different argument types or returns anything but an **id**, its function counterpart will be inadequately prototyped. Lacking a prototype, the compiler will promote **float**s to **double**s and **char**s to **int**s, which the implementation won't expect. It will therefore behave differently (and erroneously) when called as a function than when performed as a method.

To remedy this situation, it's necessary to provide your own prototype. In the example below, the declaration of the **test** variable serves to prototype the implementation of the **isEqual:** method. **test** is defined as pointer to a function that returns a BOOL and takes an **id** argument (in addition to the two "hidden" arguments). The value returned by **methodFor:** is then similarly cast to be a pointer to this same function type:

```
BOOL (*test)(id, SEL, id);
test = (BOOL (*)(id, SEL, id))[target methodFor:@selector(isEqual:)];

while ( !test(target, @selector(isEqual:), someObject) ) {
    . . .
}
```

In some cases, it might be clearer to define a type (similar to IMP) that can be used both for declaring the variable and for casting the function pointer **methodFor:** returns. The example below defines the **EqualIMP** type for just this purpose:

```
typedef BOOL (*EqualIMP)(id, SEL, id);
EqualIMP test;
test = (EqualIMP)[target methodFor:@selector(isEqual:)];

while ( !test(target, @selector(isEqual:), someObject) ) {
    . . .
}
```

Either way, it's important to cast **methodFor:**'s return value to the appropriate function type. It's not sufficient to simply call the function returned by **methodFor:** and cast the result of that call to the desired type. This can result in errors.

Note that turning a method into a function by obtaining the address of its implementation "unhides" the **self** and **_cmd** arguments.

See also: + **instanceMethodFor:**

name

– (const char *)**name**

Implemented by subclasses to return a name associated with the receiver.

By default, the string returned contains the name of the receiver's class. However, this method is commonly overridden to return a more object-specific name. You should therefore not rely on it to return the name of the class. To get the name of the class, use the class **name** method instead:

```
const char *classname = [[self class] name];
```

See also: + **name**, + **class**

notImplemented:

– **notImplemented:**(SEL)*aSelector*

Used in the body of a method definition to indicate that the programmer intended to implement the method, but left it as a stub for the time being. *aSelector* is the selector for the unimplemented method; **notImplemented:** messages are sent to **self**. For example:

```
- methodNeeded
{
    [self notImplemented:_cmd];
}
```

When a **methodNeeded** message is received, **notImplemented:** will invoke the **error:** method to generate an appropriate error message and abort the process. (In this example, **_cmd** refers to the **methodNeeded** selector.)

See also: – **subclassResponsibility:**, – **error:**

perform:

– **perform:**(SEL)*aSelector*

Sends an *aSelector* message to the receiver and returns the result of the message. This is equivalent to sending an *aSelector* message directly to the receiver. For example, all three of the following messages do the same thing:

```
id myClone = [anObject copy];
id myClone = [anObject perform:@selector(copy)];
id myClone = [anObject perform:sel_getUid("copy")];
```

However, the **perform:** method allows you to send messages that aren't determined until run time. A variable selector can be passed as the argument:

```
SEL myMethod = findTheAppropriateSelectorForTheCurrentSituation();
[anObject perform:myMethod];
```

aSelector should identify a method that takes no arguments. If the method returns anything but an object, the return must be cast to the correct type. For example:

```
char *myClass;
myClass = (char *)[anObject perform:@selector(name)];
```

Casting generally works for pointers and for integral types that are the same size as pointers (such as **int** and **enum**). Whether it works for other integral types (such as **char**, **short**, or **long**) is machine dependent. Casting doesn't work if the return is a floating type (**float** or **double**) or a structure or union. This is because the C language doesn't permit a pointer (like **id**) to be cast to these types.

Therefore, **perform:** shouldn't be asked to perform any method that returns a floating type, structure, or union, and should be used very cautiously with methods that return integral types. An alternative is to get the address of the method implementation (using **methodFor:**) and call it as a function. For example:

```
SEL aSelector = @selector(backgroundGray);
float aGray = ( (float (*)(id, SEL))
                [anObject methodFor:aSelector] )(anObject, aSelector);
```

See also: – **perform:with:**, – **perform:with:with:**, – **methodFor:**

perform:with:

– **perform:**(SEL)*aSelector* **with:***anObject*

Sends an *aSelector* message to the receiver with *anObject* as an argument. This method is the same as **perform:**, except that you can supply an argument for the *aSelector* message. *aSelector* should identify a method that takes a single argument of type **id**.

See also: – **perform:**, – **perform:with:afterDelay:cancelPrevious:** (Application Kit Object Additions)

perform:with:with:

– **perform:**(SEL)*aSelector*
with:*anObject*
with:*anotherObject*

Sends the receiver an *aSelector* message with *anObject* and *anotherObject* as arguments. This method is the same as **perform:**, except that you can supply two arguments for the *aSelector* message. *aSelector* should identify a method that can take two arguments of type **id**.

See also: – **perform:**

performv::

– **performv:**(SEL)*aSelector* **:**(marg_list)*argFrame*

Sends the receiver an *aSelector* message with the arguments in *argFrame*. **performv::** messages are used within implementations of the **forward::** method. Both arguments, *aSelector* and *argFrame*, are identical to the arguments the run-time system passes to **forward::**. They can be taken directly from that method and passed through without change to **performv::**.

performv:: should be restricted to implementations of the **forward::** method. Because it doesn't restrict the number of arguments in the *aSelector* message or their type, it may seem like a more flexible way of sending messages than **perform:**, **perform:with:**, or **perform:with:with:**. However, it's not an appropriate substitute for those methods. First, it's more expensive than they are. The run-time system must parse the arguments in *argFrame* based on information stored for *aSelector*. Second, in future releases, **performv::** may not work in contexts other than the **forward::** method.

See also: – **forward::**, – **perform:**

printForDebugger:

– (void)**printForDebugger:**(NXStream *)*stream*

Implemented by subclasses to write a useful description of the receiver to *stream*. Object's default version of this method provides the class name and the hexadecimal address of the receiver, formatted as follows:

<*classname*: 0x*address*>

Debuggers can use this method to ask objects to identify themselves.

read:

– **read:**(NXTypedStream *)*stream*

Implemented by subclasses to read the receiver's instance variables from the typed stream *stream*. You need to implement a **read:** method for any class you create, if you want its instances (or instance of classes that inherit from it) to be archivable.

The method you implement should unarchive the instance variables defined in the class in a manner that matches they way they were archived by **write:**. In each class, the **read:** method should begin with a message to **super**:

```
- read:(NXTypedStream *)stream
{
    [super read:stream];
    /* class-specific code goes here */
    return self;
}
```

This ensures that all inherited instance variables will also be unarchived.

All implementations of the **read:** method should return **self**.

After an object has been read, it's sent an **awake** message so that it can reinitialize itself, and may also be sent a **finishUnarchiving** message.

See also: – **awake**, – **finishUnarchiving**, – **write:**

respondsTo:

– (BOOL)**respondsTo:**(SEL)*aSelector*

Returns YES if the receiver implements or inherits a method that can respond to *aSelector* messages, and NO if it doesn't. The application is responsible for determining whether a NO response should be considered an error.

Note that if the receiver is able to forward the *aSelector* message to another object, it will be able to respond to the message (albeit indirectly), even though this method returns NO.

See also: – **forward::**, + **instancesRespondTo:**

self

– **self**

Returns the receiver.

See also: + **class**

startArchiving:

– **startArchiving:**(NXTypedStream *)*stream*

Implemented by subclasses to prepare an object for being archived—that is, for being written to the typed stream *stream*. A **startArchiving:** message is sent to an object just before it's archived—but only if it implements a method that can respond. The message gives the object an opportunity to do anything necessary to get itself, or the stream, ready before a **write:** message begins the archiving process.

There's no default implementation of the **startArchiving:** method. The Object class declares the method, but doesn't define it.

See also: – **awake**, – **finishUnarchiving**, – **write:**

subclassResponsibility:

– **subclassResponsibility:**(SEL)*aSelector*

Used in an abstract class to indicate that its subclasses are expected to implement *aSelector* methods. If a subclass fails to implement the method, it will inherit it from the abstract superclass. That version of the method generates an error when it's invoked. To avoid the error, subclasses must override the superclass method.

For example, if subclasses are expected to implement **doSomething** methods, the superclass would define the method this way:

```
- doSomething
{
    [self subclassResponsibility:_cmd];
}
```

When this version of **doSomething** is invoked, **subclassResponsibility:** will—by in turn invoking Object's **error:** method—abort the process and generate an appropriate error message.

(The **_cmd** variable identifies the current method selector, just as **self** identifies the current receiver. In the example above, it identifies the selector for the **doSomething** method.)

Subclass implementations of the *aSelector* method shouldn't include messages to **super** to incorporate the superclass version. If they do, they'll also generate an error.

See also: – **doesNotRecognize:**, – **notImplemented:**, – **error:**

superclass

– **superclass**

Returns the class object for the receiver's superclass.

See also: + **superclass**

write:

– **write:**(NXTypedStream *)*stream*

Implemented by subclasses to write the receiver's instance variables to the typed stream *stream*. You need to implement a **write:** method for any class you create, if you want to be able to archive its instances (or instances of classes that inherit from it).

The method you implement should archive only the instance variables defined in the class, but should begin with a message to **super** so that all inherited instance variables will also be archived:

```
- write:(NXTypedStream *)stream
{
    [super write:stream];
    /* class-specific archiving code goes here */
    return self;
}
```

All implementations of the **write:** method should return **self**.

During the archiving process, **write:** methods may be performed twice, so they shouldn't do anything other than write instance variables to a typed stream.

See also: – **read:**, – **startArchiving:**

zone

– (NXZone *)**zone**

Returns a pointer to the zone from which the receiver was allocated. Objects created without specifying a zone are allocated from the default zone, which is returned by **NXDefaultMallocZone**().

See also: + allocFromZone:, + alloc, + copyFromZone:

Types and Constants

The **objc.h** header file defines the principal data types for the Objective C language. Because it's imported by **Object.h**, and **Object.h** is indirectly imported whenever you use an Objective C class, its definitions are always available.

Other, lesser used, types and constants are documented in Chapter 15, "Run-Time System."

Defined Types

BOOL

DECLARED IN objc/objc.h

SYNOPSIS typedef char **BOOL**;

DESCRIPTION This type carries the basic boolean distinction between YES and NO (true and false).

Class

DECLARED IN objc/objc.h

SYNOPSIS typedef struct objc_class ***Class**;

DESCRIPTION Class is the data type for class objects. The **objc_class** structure it refers to holds information compiled from the class definition; details of its contents can be found in Chapter 15, "Run-Time System."

Every object has an **isa** instance variable of this type, which enables the object to identify its class.

Class objects can also be assigned to type **id**. But just as instances of a class can be statically typed by using the class name, class objects can be more particularly typed with the Class data type.

id

DECLARED IN objc/objc.h

SYNOPSIS

```
typedef struct objc_object {
    Class isa;
} *id;
```

DESCRIPTION The **id** data type designates an Objective C object of any class. All objects, including both instances and class objects, can be assigned to this type.

IMP

DECLARED IN objc/objc.h

SYNOPSIS

```
typedef id (*IMP) (id, SEL, ...);
```

DESCRIPTION This is the data type returned by Object's **methodFor:** method to identify a method implementation. It's defined as a pointer to a function that returns an **id** and takes an object (**self**) and a selector (**_cmd**) as its first two arguments.

SEL

DECLARED IN objc/objc.h

SYNOPSIS

```
typedef struct objc_selector *SEL;
```

DESCRIPTION The SEL type identifies method selectors. Valid SEL values are assigned only by the run-time system. They are never 0.

STR

DECLARED IN objc/objc.h

SYNOPSIS typedef char ***STR**;

DESCRIPTION This type is a rarely used shorthand for a character string. It's mainly of historical interest.

Symbolic Constants

Boolean Constants

DECLARED IN objc/objc.h

SYNOPSIS

YES	(BOOL)1
NO	(BOOL)0

DESCRIPTION YES and NO are the standard values assigned to BOOL variables.

Empty Objects

DECLARED IN objc/objc.h

SYNOPSIS

nil	(id)0
Nil	(Class)0

DESCRIPTION **nil** is the common notation for a NULL object. **Nil** is sometimes used for a NULL class object, but **nil** typically serves this purpose as well.

2 *Application Kit*

2 *Application Kit*

Library: libNeXT_s.a

Header File Directory: /NextDeveloper/Headers/appkit

Import: appkit/appkit.h

Introduction

The Application Kit defines a set of Objective C classes and protocols, C functions, and assorted constants and data types that are used by virtually every NeXTSTEP application. The pith of the Kit are the tools it provides for implementing a graphical, event-driven user interface:

- The Application Kit provides classes—most notably Window and View—that make drawing on the screen exquisitely succinct. Much of the unromantic work that's involved in drawing—communicating with hardware devices and screen buffers, clearing areas of the screen before drawing, coordinating overlapping drawing areas—is taken care of for you, letting you concentrate on the much more gratifying task of supplying code that simply draws. And even this task is assisted by many of the other classes and a number of C functions that provide drawing code for you.

- The Application Kit makes event handling extremely simple. The Responder class, from which many of the Kit's classes inherit, defines a mechanism by which the user's actions are passed to the objects in your application that can best respond to them. The Application class, which inherits from Responder, establishes the low-level connections that makes this system possible. It provides methods that inform your application of watershed events, such as when the user makes the application active and inactive, and when the user logs out or turns off the computer.

By using these tools, you bless your application with a look and feel that's similar to other applications, making it easier for the user to recognize and use.

Application Kit Classes and Protocols

The Application Kit is large; it comprises more than 50 classes and protocols. Figure 1 shows the inheritance hierarchy of the Application Kit classes. The following sections briefly describe the topics that the Kit addresses through its classes and protocols. Within the descriptions, class and protocol names are highlighted as they're introduced for easy identification.

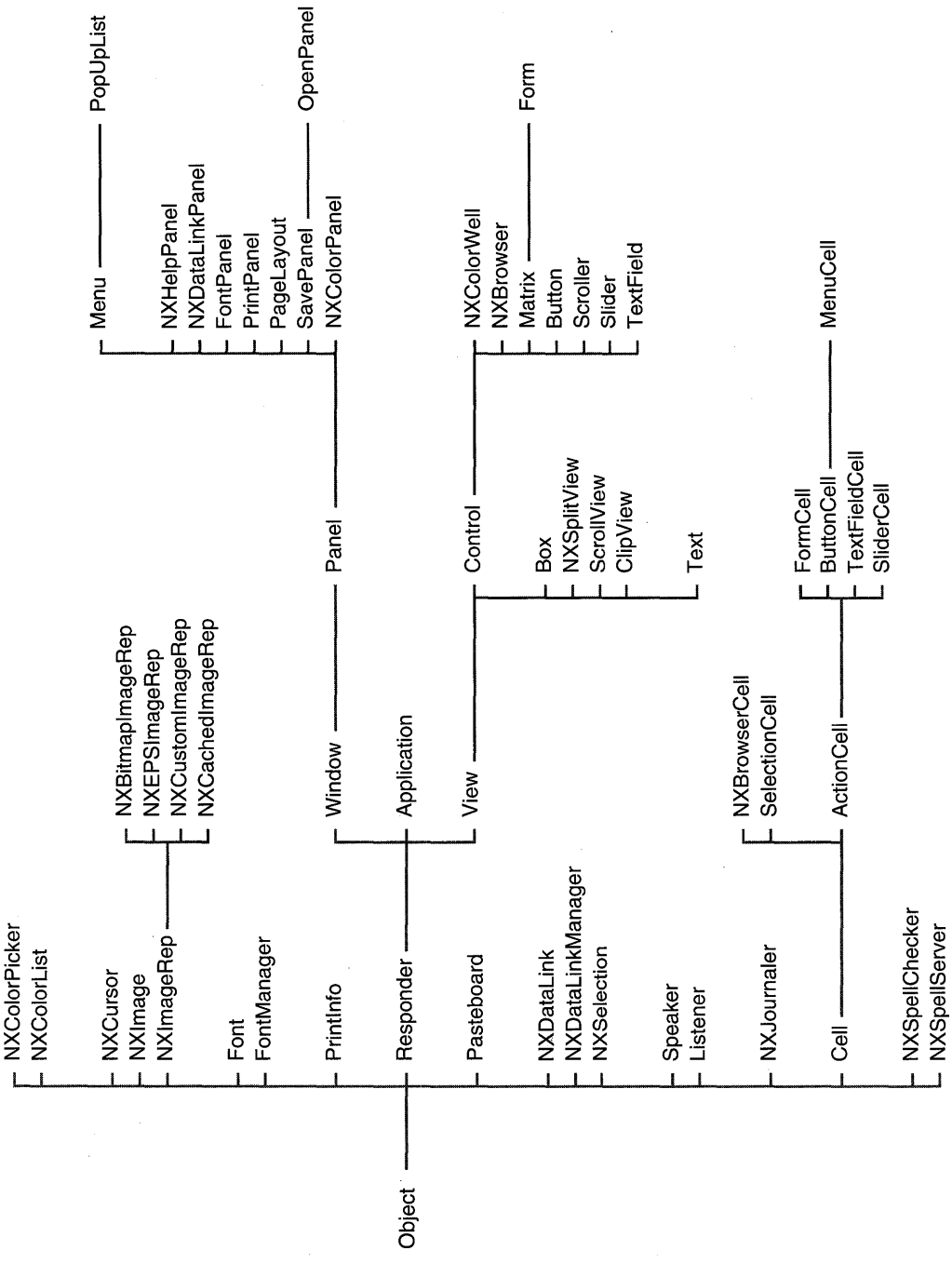


Figure 2-1. Application Kit classes

Encapsulating an Application

The central class of the Application Kit is **Application**. Every application that uses the Application Kit is given a single Application object, known to your program as NXApp, that keeps track of the application's windows and menus, controls the main event loop, lets you open NeXT Interface Builder files (with support from the **NXNibNotification** protocol), and maintains information regarding printing, languages, screens, color support, and so on.

General Drawing and Event Handling

The **Window** and **View** classes are the centerpieces of drawing. More specifically, Windows represent rectangular areas on the screen in which the user works. To the extent that everything the user does is directed to a Window, an application's set of Windows *is* the application. Views are areas within Windows that perform your application's drawing.

Panel is a subclass of Window that you use to display transient, global, or pressing information. For example, you would use a Panel, rather than an instance of Window, to display error messages, or to query the user for a response to remarkable or unusual circumstances.

The **Responder** class defines the *responder chain*, an ordered list of objects that respond to user events. When the user clicks the mouse or presses a key, an event is generated and passed up the responder chain in search of an object that can respond to it.

Menus and Cursors

The **Menu**, **MenuCell**, and **NXCursor** classes define the look and behavior of the menus and cursors that your application displays to the user.

Grouping and Scrolling Views

The **Box**, **ScrollView**, and **NXSplitView** classes provide graphic accoutrements to some other View or collection of Views. A Box groups some number of other Views, and lets you draw a border around the entire group. NXSplitView lets you "stack" Views vertically, apportioning to each View some amount of a common territory; a sliding control bar lets the user redistribute the territory among Views. ScrollView, and its helper **ClipView**, provide a scrolling mechanism as well as the graphic objects that let the user initiate and control a scroll.

Controlling an Application

The **Control** and **Cell** classes, and their subclasses, define an easily recognized set of buttons, sliders, and browsers that the user can manipulate graphically to control some aspect of your application. Just what a particular control affects is up to you: When a control is "touched," it sends a certain message to a specific object. This is the *targeted action paradigm*; for each Control, you define both the target (an object) and the action (the message that's sent to that object).

A Cell completes the implementation of a Control. In general, for each Control there is a corresponding Cell; thus a button comprises a **Button** and a **ButtonCell**, a slider is a **Slider** and **SliderCell**, and so on.

Text and Fonts

Most applications display text in some form. The **Text** and **TextField** classes make this presentation as straightforward and simple as possible. The size of the Text class is daunting at first, but for simple text presentation only a handful of methods are actually needed (or you can use the streamlined TextField class). More complicated text-based applications, such as word processors, can take advantage of the Text class' more sophisticated features, such as rulers and break tables.

The **Font** and **FontManager** encapsulate and manage different font families, sizes, and variations. The Font class defines a single object for each distinct font; for efficiency, these objects, which can be rather large, are shared by all the objects in your application.

The **FontPanel** class defines the font-specification panel that's presented to the user.

Graphics and Color

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

Color is incorporated and supported by **NXColorPanel**, **NXColorList**, **NXColorPicker**, and **NXColorWell**. These are mostly interface classes: They define and present Panels and Views that allow the user to select and apply colors. The **NXColorPicking** protocol lets you extend the standard Colors panel.

The four standard color formats—RGB, CMYK, HSB, and grayscale—are recognized by the color classes. You can also tell the classes to recognize custom representations.

Printing and Faxing

The **NXPrinter**, **PrintPanel**, **PageLayout**, and **PrintInfo** classes work together to provide the means for printing and faxing the information that your application displays in its Windows and Views. For more control, the View and Window classes and the **NXPrintingUserInterface** protocol define methods that can fine-tune the printing and faxing mechanism.

Accessing the File System

The Application Kit doesn't provide a class that defines objects to correspond to files on the disk. However, the **OpenPanel** and **SavePanel** provide a convenient and familiar user interface to the file system.

Sharing Data with Other Applications

The **Pasteboard** class defines a repository for data that's copied from your application, making this data available to any application that cares to use it. This is the familiar cut-copy-paste mechanism. The **NXServicesRequest** protocol uses the Pasteboard to communicate data that's passed between applications by a registered service.

The **Listener** and **Speaker** classes provide a more specific communication between separate applications in which one application (using a Speaker) provides data to which the other (through a Listener) is programmed to respond.

Finally, an intimate link between applications can be created through the **NXDataLink**, **NXDataLinkManager**, **NXDataLinkPanel**, and **NXSelection** classes. Through these classes, multiple applications can share the same data. A change to the data in one application is seen immediately in all others that display that data.

Spell-Checking

The **NXSpellServer** class lets you define a spell-checking facility and provide it as a service to other applications. To connect your application to a spelling checker, you use the **NXSpellChecker** class. The **NXSelectText**, **NXIgnoreMisspelledWords**, and **NXChangeSpelling** protocols support the spell-checking mechanism.

Journaling and Help

The **NXJournaler** class provides an interactive recording and playback environment in which you can run your application. During recording, events are noted, time-stamped, and stored. The journaled "script" can then be played back; your application will run itself to the delight of the assembled throng.

The **NXHelpPanel** class is the central component of the NeXTSTEP help system. It provides a panel that displays the text and illustrations that constitute your application's help information, and it associates user-interface objects with specific passages of that text.

Application Kit Functions

The "Functions" section, later in this chapter, describes the functions (and function-like macros) that are provided by the Application Kit. Many of the functions are auxiliary to the Kit's classes in that they augment or are superceded by one or more classes. Of the rest, some functions provide information or functionality that can't be gotten elsewhere, while some others are convenient but not necessarily the only way to address a particular topic.

The following sections don't attempt to describe what individual functions do—the names of the functions are fairly descriptive in themselves—they merely list the functions as they fall into broad categories.

Drawing and Graphic Geometry

These functions draw standard interface accoutrements, or examine and manipulate graphic regions.

- NXDrawButton(), NXDrawGrayBezel(), NXDrawGroove(), NXDrawWhiteBezel(), NXDrawTiledRects(), NXFrameRect(), NXFrameRectWithWidth()
- NXAttachPopUpList(), NXCreatePopUpListButton()
- NXRectClip(), NXRectClipList(), NXRectFill(), NXRectFillList(), NXRectFillListWithGrays(), NXEraseRect(), NXHighlightRect()
- NXSetRect(), NXOffsetRect(), NXInsetRect(), NXIntegralRect(), NXDivideRect()
- NXMouseInRect(), NXPointInRect(), NXIntersectsRect(), NXContainsRect(), NXEqualRect(), NXEmptyRect()
- NXUnionRect(), NXIntersectionRect()

- NX_X(), NX_Y(), NX_WIDTH(), NX_HEIGHT(), NX_MAXX(), NX_MAXY(), NX_MIDX(), NX_MIDY()
- NXFindPaperSize()

Images

These functions access image data (note, however, that they're superceded by NXImage and related classes).

- NXCopyBits()
- NXCopyBitmapFromGstate()

Colors

Since there isn't a class that represents individual colors, these function are indispensable for dealing with color.

- NXSetColor()
- NXColorListName(), NXColorName(), NXFindColorNamed()
- NXReadPixel()
- NXEqualColor()
- NXChangeRedComponent(), NXChangeGreenComponent(), NXChangeBlueComponent(), NXChangeCyanComponent(), NXChangeMagentaComponent(), NXChangeYellowComponent(), NXChangeBlackComponent(), NXChangeHueComponent(), NXChangeSaturationComponent(), NXChangeBrightnessComponent(), NXChangeGrayComponent(), NXChangeAlphaComponent()
- NXConvertColorToRGBA(), NXConvertColorToCMYKA(), NXConvertColorToHSBA(), NXConvertColorToGrayAlpha(), NXConvertColorToRGB(), NXConvertColorToCMYK(), NXConvertColorToHSB(), NXConvertColorToGray()
- NXConvertRGBAToColor(), NXConvertCMYKAToColor(), NXConvertHSBAToColor(), NXConvertGrayAlphaToColor(), NXConvertRGBToColor(), NXConvertCMYKToColor(), NXConvertHSBToColor(), NXConvertGrayToColor()

- NXRedComponent(), NXGreenComponent(), NXBlueComponent(), NXCyanComponent(), NXMagentaComponent(), NXYellowComponent(), NXBlackComponent(), NXHueComponent(), NXSaturationComponent(), NXBrightnessComponent(), NXGrayComponent(), NXAlphaComponent()

Text, Fonts, and Characters

These functions let you query and manipulate various aspects of displayed text.

- NXReadWordTable(), NXWriteWordTable()
- NXScanALine(), NXDrawALine()
- NXFieldFilter(), NXEditorFilter()
- NXTextFontInfo()
- NXOrderStrings(), NXDefaultStringOrderTable()

Windows and Screen Devices

Through these functions you can access the Window Server's windows (the devices that underlie Window objects) and retrieve information that aids in matching a Window object to the attributes of the screen upon which it's placed.

- NXColorSpaceFromDepth(), NXBPSFromDepth(), NXNumberOfColorComponents(), NXGetBestDepth()
- NXConvertWinNumToGlobal(), NXConvertGlobalToWinNum()
- NXCountWindows(), NXWindowList()
- NXGetWindowServerMemory()
- NXSetGState(), NXCopyCurrentGState()

Attention Panels

Attention panels are much easier to create through the following functions rather than by creating individual Panel objects.

- NXRunAlertPanel(), NXRunLocalizedAlertPanel(), NXGetAlertPanel(), NXFreeAlertPanel()

Events

These functions let you query for events and provide some control over the events that your application manufactures.

- NXGetOrPeekEvent()
- NXUserAborted(), NXResetUserAbort()
- NXBeginTimer(), NXEndTimer()
- NXJournalMouse()
- NXPing()

The File System and Operating Environment

These functions provide information about the user, manipulate file names, and play the system beep.

- NXHomeDirectory(), NXUserName()
- NXCompleteFilename()
- NXExpandFilename()
- NXBeep()

Pasteboard Functions

These functions access data on the pasteboard:

- NXCreateFileContentsPboardType(), NXCreateFilenamePboardType()
- NXGetFileType(), NXGetFileTypes()
- NXReadColorFromPasteboard(), NXWriteColorToPasteboard()

Archiving

The archiving functions let you read and write individual items (rather than entire objects) from and to files.

- NXReadPoint(), NXWritePoint(), NXReadRect(), NXWriteRect(), NXReadSize(), NXWriteSize()
- NXReadColor(), NXWriteColor()

Named Objects

These functions let you refer to objects by name.

- NXGetNamedObject(), NXGetObjectName(), NXNameObject(), NXUnnameObject()

Services, Data Links, and Remote Messages

These functions assist the services system, data links, and aid in getting data into and from a remote message (a message passed between applications).

- NXSetServicesMenuItemEnabled(), NXIsServicesMenuItemEnabled()
- NXUpdateDynamicServices()
- NXPerformService()
- NXFrameLinkRect(), NXLinkFrameThickness()
- NXCopyInputData(), NXCopyOutputData()
- NXRemoteMethodFromSel(), NXResponsibleDelegate()
- NXPortFromName(), NXPortNameLookup()

Error Handling and Debugging

These functions help you respond to errors and to debug your application.

- NXDefaultTopLevelErrorHandler(), NXSetTopLevelErrorHandler(), NXTopLevelErrorHandler()
- NXLogError()
- NXRegisterErrorReporter(), NXRemoveErrorReporter(), NXReportError()
- NX_ASSERT()
- NX_PSDEBUG

Allocating Memory

These functions let you allocate and free memory. The "chunk" functions are used, principally, by the Text class.

- NX_MALLOC(), NX_REALLOC(), NX_FREE()
- NX_ZONEMALLOC(), NX_ZONEREALLOC()
- NXChunkMalloc(), NXChunkRealloc(), NXChunkGrow(), NXChunkCopy(), NXChunkZoneMalloc(), NXChunkZoneRealloc(), NXChunkZoneGrow(), NXChunkZoneCopy()

Classes

ActionCell

Inherits From: Cell : Object

Declared In: appkit/ActionCell.h

Class Description

An ActionCell defines an active area inside a Control (an instance of Control or one of its subclasses). As a Control's active area, an ActionCell does three things: it performs display of text or an icon; it provides the Control with a target and an action; and it handles mouse (cursor) tracking by properly highlighting its area and sending action messages to its target based on cursor movement. You can set an ActionCell's Control only by sending the **drawSelf:inView:** message to the ActionCell, passing the Control as the argument for the **inView:** keyword of the method.

ActionCell implements the target object and action method as defined by its superclass, Cell. As a user manipulates a Control, ActionCell's **trackMouse:inRect:ofView:** method (inherited from Cell) updates its appearance and sends the action message to the target object with the Control object as the only argument.

A single Control may have more than one ActionCell. An integer tag is used to identify an ActionCell; this is important for Controls that contain more than one ActionCell. Note, however, that no checking is done by the ActionCell object itself to ensure that the tag is unique. See the Matrix class for an example of a subclass of Control that contains multiple ActionCells.

Many of the methods that define the contents and look of an ActionCell, such as **setFont:** and **setBordered:**, are reimplementations of methods inherited from Cell. They're subclassed to ensure that the ActionCell is redisplayed if it's currently in a Control.

Instance Variables

int **tag**;
id **target**;
SEL **action**;

tag	An integer used to identify the ActionCell.
target	The object that is sent the ActionCell's action.
action	The message that the ActionCell sends to its target.

Method Types

Configuring an ActionCell	– setEnabled: – setBezeled: – setBordered: – setAlignment: – setFloatingPointFormat:left:right: – setFont: – setIcon:
Manipulating ActionCell values	– doubleValue – floatValue – intValue – setStringValue: – setStringValueNoCopy:shouldFree: – stringValue
Displaying	– drawSelf:inView: – controlView
Target and action	– setAction: – action – setTarget: – target
Assigning a tag	– setTag: – tag
Archiving	– write: – read:

Instance Methods

action

– (SEL)**action**

Returns the ActionCell's action method. Keep in mind that the argument of an action method sent by an ActionCell is its associated Control (the object returned by **controlView**).

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

controlView

– **controlView**

Returns the Control in which the ActionCell was most recently drawn. In general, your code should use the object returned by this method only to (indirectly) redisplay the ActionCell. For example, the subclasses of ActionCell defined by the Application Kit invoke this method in order to send the Control a message such as **updateCellInside:**.

The Control in which an ActionCell is drawn is set automatically by the **drawSelf:inView:** method. You can't explicitly set the Control.

See also: – **drawSelf:inView:**

doubleValue

– (double)**doubleValue**

Returns the ActionCell's contents as a double-precision floating point number. If the ActionCell is being edited when this message is received, editing is validated first.

See also: – **setDoubleValue:** (Cell), – **floatValue**, – **intValue**, – **stringValue**, – **validateEditing** (Control)

drawSelf:inView:

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

Displays the ActionCell. Sets the ActionCell's Control to *controlView* and performs drawing if and only if *controlView* is a Control object (an instance of Control or a subclass thereof). You must lock focus on the Control before invoking this method (Control's **display** method automatically performs this).

See also: – **drawSelf:inView:** (Cell)

floatValue

– (float)**floatValue**

Returns the ActionCell's contents as a single-precision floating point number. If the ActionCell is being edited when this message is received, editing is validated first.

See also: – **setFloatValue:** (Cell), – **doubleValue**, – **intValue**, – **stringValue**, – **validateEditing** (Control)

intValue

– (int)**intValue**

Returns the ActionCell's contents as an integer. If the ActionCell is being edited when this message is received, editing is validated first.

See also: – **setIntValue:** (Cell), – **doubleValue**, – **floatValue**, – **stringValue**, – **validateEditing** (Control)

read:

– **read:**(NXTypedStream *)*stream*

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

See also: – **write:**

setAction:

– **setAction:**(SEL)*aSelector*

Sets the ActionCell's action method to *aSelector*. The argument of an action method sent by an ActionCell is its associated Control (the object returned by **controlView**). Returns **self**.

See also: – **action**, – **setTarget:**, – **controlView**, – **sendAction:to:** (Control)

setAlignment:

– **setAlignment:**(int)*mode*

If the ActionCell is a text Cell (type NX_TEXTCELL), this sets its text alignment to *mode*, which should be NX_LEFTALIGNED, NX_CENTERED, or NX_RIGHTALIGNED. If it's currently in a Control view, the ActionCell is redisplayed or marked as needing redisplay. Returns **self**.

See also: – **alignment** (Cell)

setBezeled:

– **setBezeled:**(BOOL)*flag*

Adds or removes the ActionCell's bezel, according to the value of *flag*. Adding a bezel will remove the ActionCell's border, if any. If it's currently in a Control view, the ActionCell is redisplayed or marked as needing redisplay. Returns **self**.

See also: – **isBezeled** (Cell), – **setBordered:**

setBordered:

– **setBordered:**(BOOL)*flag*

Adds or removes the ActionCell's border, according to the value of *flag*. The border is black and has a width of 1.0. Adding a border will remove the ActionCell's bezel, if any. If it's currently in a Control view, the ActionCell is redisplayed or marked as needing redisplay. Returns **self**.

See also: – **isBordered** (Cell), – **setBezeled:**

setEnabled:

– **setEnabled:**(BOOL)*flag*

Enables or disables the ActionCell's ability to receive mouse events, according to the value of *flag*. If it's currently in a Control view, the ActionCell is redisplayed or marked as needing redisplay. Returns **self**.

See also: – **isEnabled** (Cell)

setFloatingPointFormat:left:right:

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

Sets the ActionCell's floating point format as described in the Cell class specification for the **setFloatingPointFormat:left:right:** method. If it's currently in a Control view, the ActionCell is redisplayed or marked as needing redisplay. Returns **self**.

See also: – **setFloatingPointFormat:left:right:** (Cell)

setFont:

– **setFont:***fontObject*

If the ActionCell is a text Cell (type NX_TEXTCELL), this sets its Font to *fontObject*. In addition, if it's currently in a Control view, the ActionCell is redisplayed or marked as needing redisplay. Returns **self**.

See also: – **font** (Cell)

setIcon:

– **setIcon:**(const char *)*iconName*

Sets the ActionCell's icon to *iconName* and sets its Cell type to NX_ICONCELL. If it's currently in a Control view, the ActionCell is redisplayed or marked as needing redisplay. Returns **self**.

See also: – **setIcon:** (Cell)

setStringValue:

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

Sets the ActionCell's contents to a copy of *aString*. If it's currently in a Control view, the ActionCell is redisplayed or marked as needing redisplay. Returns **self**.

See also: – **setStringValueNoCopy:shouldFree:**, – **setStringValue:** (Cell), – **stringValue**, – **doubleValue**, – **floatValue**, – **intValue**

setStringValueNoCopy:shouldFree:

– **setStringValueNoCopy:**(char *)*aString* **shouldFree:**(BOOL)*flag*

Sets the ActionCell's contents to a *aString*. If *flag* is YES, *aString* will be freed when the ActionCell is freed. If it's currently in a Control view, the ActionCell is redisplayed or marked as needing redisplay. Returns **self**.

See also: – **setStringValue:**, – **setStringValueNoCopy:shouldFree:** (Cell), – **stringValue**, – **doubleValue**, – **floatValue**, – **intValue**

setTag:

– **setTag:**(int)*anInt*

Sets the ActionCell's tag to *anInt*. The tag can be used to identify the ActionCell in a Control that contains multiple Cells (Matrix, for example). Returns **self**.

See also: – **tag**, – **setTag:** (Control)

setTarget:

– **setTarget:***anObject*

Sets the ActionCell's target to *anObject*. This is the object that is sent the ActionCell's action method. Returns **self**.

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

stringValue

– (const char *)**stringValue**

Returns the ActionCell's contents as a string. If the ActionCell is being edited when this message is received, editing is validated first.

See also: – **setStringValue:**, – **stringValue** (Cell), – **validateEditing** (Control), – **doubleValue**, – **floatValue**, – **intValue**

tag

– (int)**tag**

Returns the ActionCell's tag. The tag can be used to identify the ActionCell in a Control that contains multiple Cells (Matrix, for example).

See also: – **setTag:**, – **tag** (Control)

target

– **target**

Returns the ActionCell's target, the object that is sent the ActionCell's action method.

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

write:

– **write:**(NXTypedStream *)*stream*

Writes the ActionCell to the typed stream *stream*. Returns **self**.

See also: – **read:**

Application

Inherits From: Responder : Object

Declared In: appkit/Application.h

Class Description

The Application class provides the central framework of your application's execution. Every application must have exactly one object that is an instance of Application (or of a custom subclass of Application). Project Builder automatically inserts into the main file (the file that contains the **main()** function) code that creates that object and stores it as the global variable NXApp. The automatically generated code then loads your application's nib file, and starts the event loop by sending a **run** message to NXApp.

Creating the Application object connects the program to the Window Server and initializes its PostScript environment. The Application object maintains a list of all the Windows that the application uses, so it can retrieve any of the application's Views.

The Application object's main task is to receive events from the Window Server and distribute them to the proper Responders. The Application object handles a system event itself. It translates a Window event into a message forwarded to the affected Window object. A key-down event that occurs while the Command key is pressed is translated into a **commandKey:** message, and every Window has an opportunity to respond to it. Other keyboard and mouse events are sent to the Window associated with the event; the Window then distributes them to the objects in its View hierarchy.

In general, it's neater and cleaner to separate the code that embodies your program's functionality into a number of custom objects. Usually those custom objects are subclasses of Object. Methods defined in your custom objects can be invoked from a small dispatcher object without being closely tied to the Application object. It is rarely necessary to create a custom subclass of Application. You will need to do so only if you need to provide your own special response to messages that are routinely sent to the Application object. If you do create a custom subclass of Application, it's the object representing your custom class that gets the name NXApp and receives the **run** message.

The Application object can be assigned a delegate that responds on its behalf to notification messages addressed to the Application object. For a few of these notification methods, if you have created a subclass of Application and it implements the method but the delegate doesn't, that message is sent to **self** (and thus to a subclass method). Where that is true, it's noted in the method's description in "Methods Implemented by the Delegate," below.

Since an application must have one and only one Application object, you must use **new** to create it. You can't use **alloc**, **allocFromZone:**, or **init** to create or initialize an Application object.

When your application is launched, its main nib file (if it has one) is loaded; the objects stored in the nib file are unarchived. When unarchived, each gets an **awake** message and then a **finishUnarchiving** message. Note that some objects in the nib file—for example, objects represented by the proxy CustomView object—are simply referenced, not archived. Those objects don't get the **awake** or **finishUnarchiving** messages. Instead, they're instantiated through the **alloc** and **init** mechanism.

Instance Variables

```
char *appName;
NXEvent currentEvent;
id windowList;
id keyWindow;
id mainWindow;
id delegate;
int *hiddenList;
int hiddenCount;
const char *hostName;
DPSContext context;
int contextNum;
id appListener;
id appSpeaker;
port_t replyPort;
NXSize screenSize;
short running;
struct __appFlags {
    unsigned int hidden:1;
    unsigned int autoupdate:1;
    unsigned int active:1;
} appFlags;
```

appName	The name of your application; used by the defaults system and the application's Listener object
currentEvent	The event most recently retrieved from the event queue

windowList	A List containing all the Windows to which the Application has access
keyWindow	The Window that receives keyboard events
mainWindow	The Window that receives menu commands and action messages from a Panel
delegate	The object that responds to notification messages
hiddenList	The List of Windows belonging to the Application at the time the Application was hidden
hiddenCount	The number of Windows referred to by **hiddenList**
hostName	The name of the machine running the Window Server
context	The Display PostScript context connected to the Window Server
contextNum	A number identifying the application's Display PostScript context
appListener	The Application object's Listener
appSpeaker	The Application object's Speaker
replyPort	A general purpose reply port for the Application object's Speakers
screenSize	The size of the screen that this application is running on
running	The nested level of **run** and **runModalFor:**
appFlags.hidden	YES if the Application's Windows are currently hidden
appFlags.autoupdate	YES if the Application object is to send an **update** message to each Window after an event has been processed
appFlags.active	YES if the Application is the active application

Method Types

Initializing the class	+ initialize + alloc + allocFromZone:
Creating and freeing instances	+ new – free

Setting up the application	+ workspace – loadNibFile:owner: – loadNibFile:owner:withNames: – loadNibFile:owner:withNames:fromZone: – loadNibSection:owner: – loadNibSection:owner:withNames: – loadNibSection:owner:withNames:fromHeader: – loadNibSection:owner:withNames:fromZone: – loadNibSection:owner:withNames:fromHeader: fromZone: – appName – setMainMenu: – mainMenu
Responding to notification	– applicationWillLaunch: – applicationDidLaunch: – applicationDidTerminate:
Changing the active application	– activeApp – becomeActiveApp – activate: – activateSelf: – isActive – resignActiveApp – deactivateSelf
Running the event loop	– run – isRunning – stop: – runModalFor: – stopModal – stopModal: – abortModal – beginModalSession:for: – runModalSession: – endModalSession: – delayedFree: – sendEvent:

Getting and peeking at events	– currentEvent – getNextEvent: – getNextEvent:waitFor:threshold: – peekAndGetNextEvent: – peekNextEvent:into: – peekNextEvent:into:waitFor:threshold:
Journaling	– setJournalable: – isJournalable – masterJournaler – slaveJournaler
Handling user actions and events	– applicationDefined: – hide: – isHidden – unhide – unhide: – unhideWithoutActivation: – powerOff: – powerOffIn:andSave: – rightMouseDown: – unmounting:ok:
Sending action messages	– sendAction:to:from: – tryToPerform:with: – calcTargetForAction:
Remote messaging	– setAppListener: – appListener – setAppSpeaker: – appSpeaker – appListenerPortName – replyPort
Managing Windows	– appIcon – findWindow: – getWindowNumbers:count: – keyWindow – mainWindow – makeWindowsPerform:inOrder: – setAutoupdate: – updateWindows – windowList – miniaturizeAll: – preventWindowOrdering

Managing the Windows menu	– setWindowsMenu: – windowsMenu – arrangeInFront: – addWindowsItem:title:filename: – changeWindowsItem:title:filename: – removeWindowsItem: – updateWindowsItem:
Managing Panels	– showHelpPanel: – orderFrontDataLinkPanel:
Managing the Services menu	– setServicesMenu: – servicesMenu – registerServicesMenuSendTypes:andReturnTypes: – validRequestorForSendType:andReturnType:
Managing screens	– mainScreen – colorScreen – getScreens:count: – getScreenSize:
Querying the application	– context – focusView – hostName
Reporting current languages	– systemLanguages
Using files	– openFile:ok: – openTempFile:ok: – fileOperationCompleted:
Responding to devices	– mounted: – unmounted:
Printing	– setPrintInfo: – printInfo – runPageLayout:
Color	– orderFrontColorPanel: – setImportAlpha: – doesImportAlpha
Terminating the application	– terminate:
Assigning a delegate	– setDelegate: – delegate

Class Methods

alloc

This method cannot be used to create an Application object. Use **new** instead. The method is implemented only to prevent you from using it; if you do use it, it generates an error message.

See also: + **new**

allocFromZone:

This method cannot be used to create an Application object. Use **new** instead. The method is implemented only to prevent you from using it; if you do use it, it generates an error message.

See also: + **new**

initialize

+ **initialize**

Registers defaults used by the Application class. You never send this message directly; it's sent for you when your application starts. Returns **self**.

new

+ **new**

Creates a new Application object and assigns it to the global variable NXApp. A program can have only one Application object, so this method just returns NXApp if the Application object already exists. This method also makes a connection to the Window Server, loads the PostScript procedures the application needs, and completes other initialization. Your program should generally invoke this method as one of the first statements in **main**(); this is done for you if you create your application with Interface Builder. Returns the Application object.

See also: – **run**

workspace

+ (id <NXWorkspaceRequestProtocol>)**workspace**

Returns an object representing the Workspace Manager. Your code can send it a message asking it to do such things as open a file. The Workspace Manager responds to the NXWorkspaceRequest protocol. Here's an example of asking the Workspace Manager for the icon for the file **x.draw**:

```
NXImage *i = [[Application workspace] getIconForFile:"x.draw"];
```

Instance Methods

abortModal

– (void)**abortModal**

Aborts the modal event loop by raising the NX_abortModal exception, which is caught by **runModalFor:**, the method that started the modal loop. Since this method raises an exception, it never returns; **runModalFor:**, when stopped with this method, returns NX_RUNABORTED. This method is typically invoked from procedures registered with **DPSAddTimedEntry**(), **DPSAddPort**(), or **DPSAddFD**(). Note that you can't use this method to abort modal sessions, where you control the modal loop and periodically invoke **runModalSession:**.

See also: – **runModalFor:**, – **runModalSession:**, – **endModalSession:**, – **stopModal**, – **stopModal:**

activate:

– (int)**activate:**(int)*contextNumber*

Makes the application identified by *contextNumber* the active application. The argument *contextNumber* is the PostScript context number of the application to be activated. Normally, you shouldn't invoke this method; the Application Kit is responsible for proper activation. Returns the PostScript context number of application that was previously active.

See also: – **isActive**, – **activateSelf:**, – **deactivateSelf**

activateSelf:

– (int)**activateSelf:**(BOOL)*flag*

Makes the receiving application the active application. If *flag* is NO, the application is activated only if no other application is currently active. Normally, this method is invoked with *flag* set to NO. When the Workspace Manager launches an application, it deactivates itself, so **activateSelf:**NO allows the application to become active if the user waits for it to launch, but the application remains unobtrusive if the user activates another application. If *flag* is YES, the application will always activate. Regardless of the setting of *flag*, there may be a time lag before the application activates; you should not assume that the application will be active immediately after sending this message.

Note that you can make one of your Windows the key window without changing the active application; when you send a **makeKeyWindow** message to a Window, you simply ensure that the Window will be the key window when the application is active.

You should rarely need to invoke this method. Under most circumstances the Application Kit takes care of proper activation. However, you might find this method useful if you implement your own methods for interapplication communication. This method returns the PostScript context number of the previously active application.

See also: – **activeApp**, – **activate:**, – **deactivateSelf**, – **makeKeyWindow** (Window)

activeApp

– (int)**activeApp**

Returns the active application's PostScript context number. If no application is active, returns zero.

See also: – **isActive**, – **activate:**

addWindowsItem:title:filename:

– **addWindowsItem:***aWindow*
title:(const char *)*aString*
filename:(BOOL)*isFilename*

Adds an item to the Windows menu corresponding to the Window *aWindow*. If *isFilename* is NO, *aString* appears literally in the menu. If *isFilename* is YES, *aString* is assumed to be a converted name with the name of the file preceding the path (the way Window's

setTitleAsFilename: method shows a title). If an item for *aWindow* already exists in the Windows menu, this method has no effect. You rarely invoke this method because an item is placed in the Windows menu for you whenever a Window's title is set. Returns **self**.

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

appIcon

– **appIcon**

Returns the Window object that represents the application in the Workspace Manager (containing the application's title and icon).

applicationDefined:

– **applicationDefined:**(NXEvent *)*theEvent*

Invoked when the application receives an application-defined (NX_APPDEFINED) event. This is a vehicle in which you provide whatever response you want, by overriding the default definition in a subclass or defining this method in the delegate. Returns **self**.

applicationDidLaunch:

– (int)**applicationDidLaunch:**(const char *)*appName*

Notification from the Workspace Manager that the application whose name is *appName* has launched. This is one of the messages the Application will receive if it has previously sent the Workspace Manager the message **beginListeningForApplicationStatusChanges**.

If the delegate implements the method **app:applicationDidLaunch:**, that message is sent to it. If the delegate doesn't implement it, the method is handled by the Application subclass object (if you created one). The return is an arbitrary integer; your application defines and interprets it. If you neither provide a delegate method nor override in a subclass, the default definition simply returns 0.

See also: – **app:applicationDidLaunch:** (Application delegate method), – **beginListeningForApplicationStatusChanges** (NXWorkspaceRequest protocol)

applicationDidTerminate:

– (int)**applicationDidTerminate:**(const char *)*appName*

Notification from the Workspace Manager that the application whose name is *appName* has terminated. This is one of the messages the Application will receive if it has previously sent the Workspace Manager the message **beginListeningForApplicationStatusChanges**.

If the delegate implements the method **app:applicationDidTerminate:**, that message is sent to it. If the delegate doesn't implement it, the method is handled by the Application subclass object (if you created one). The return is an arbitrary integer; your application defines and interprets it. If you neither provide a delegate method nor override in a subclass, the default definition simply returns 0.

See also: – **app:applicationDidTerminate:** (Application delegate method), – **beginListeningForApplicationStatusChange**s (NXWorkspaceRequest protocol)

applicationWillLaunch:

– (int)**applicationWillLaunch:**(const char *)*appName*

Notification from the Workspace Manager that the application whose name is *appName* is about to launch. This is one of the messages the Application will receive if it has previously sent the Workspace Manager the message **beginListeningForApplicationStatusChanges**.

If the delegate implements the method **app:applicationWillLaunch:**, that message is sent to it. If the delegate doesn't implement it, the method is handled by the Application subclass object (if you created one). The return is an arbitrary integer; your application defines and interprets it. If you neither provide a delegate method nor override in a subclass, the default definition simply returns 0.

See also: – **app:applicationWillLaunch:** (Application delegate method), – **beginListeningForApplicationStatusChange**s (NXWorkspaceRequest protocol)

appListener

– **appListener**

Returns the Application object's Listener—the object that will receive messages sent to the port that's registered for the application's name. If you don't send a **setAppListener:** message before your application starts running, an instance of Listener is created for you. (Note, however, that to communicate with the Workspace Manager to do such things as open files, you should send messages to the object that represents the Workspace Manager,

returned by the **workspace** class method; it responds to the NXWorkspaceRequest protocol.)

See also: – **setAppListener:**, – **appListenerPortName**, – **run**, + **workspace**

appListenerPortName

– (const char *)**appListenerPortName**

Returns the name used to register the Application object's Listener. The default is the same name that's returned by the Application object's **appName** method. If a different name is desired, this method should be overridden. Messages sent by name to **appListenerPortName** will be received by your Application object.

See also: – **checkInAs:** (Listener), – **appName**, **NXPortFromName()**

appName

– (const char *)**appName**

Returns the name under which the Application object has been registered for defaults. This name is also used for messaging unless the messaging name was changed by overriding **appListenerPortName**.

See also: – **appListenerPortName**

appSpeaker

– **appSpeaker**

Returns the Application object's Speaker. You can use this object to send messages to other applications.

See also: – **setSendPort:** (Speaker)

arrangeInFront:

– **arrangeInFront:***sender*

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

See also: – **removeWindowsItem:**, – **makeKeyAndOrderFront:** (Window)

becomeActiveApp

– **becomeActiveApp**

Sends the **appDidBecomeActive:** message to the Application object's delegate. This method is invoked when the application is activated. You never send a **becomeActiveApp** message directly, but you can override this method in a subclass. Returns **self**.

See also: – **activateSelf:**, – **appDidBecomeActive:** (delegate method)

beginModalSession:for:

– (NXModalSession *)**beginModalSession:**(NXModalSession *)*session*
for:*theWindow*

Prepares the application for a modal session with *theWindow*. In other words, prepares the application so that mouse events get to it only if they occur in *theWindow*. If *session* is NULL, an NXModalSession is allocated; otherwise the given storage is used. (The sender could declare a local NXModalSession variable for this purpose.) *theWindow* is made the key window and ordered to the front.

beginModalSession:for: should be balanced by **endModalSession:**. If an exception is raised, **beginModalSession:for:** arranges for proper cleanup. Do *not* use NX_DURING constructs to send an **endModalSession:** message in the event of an exception. Returns the NXModalSession pointer that's used to refer to this session.

See also: – **runModalSession:**, – **endModalSession:**

calcTargetForAction:

– **calcTargetForAction:**(SEL)*theAction*

Returns the first object in the responder chain that responds to the message *theAction*. The message isn't actually dispatched. Note that this method doesn't test the value that the responding object would return should the message be sent; specifically, it doesn't test to see if the responder would return **nil**. Returns **nil** if no responder is found.

See also: – **sendAction:to:from:**

changeWindowsItem:title:filename:

– **changeWindowsItem:***aWindow*
title:(const char *)*aString*
filename:(BOOL)*isFilename*

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

See also: – **addWindowsItem:title:filename:**, – **setTitle:** (Window), – **setTitleAsFilename:** (Window)

colorScreen

– (const NXScreen *)**colorScreen**

Returns the screen that can best represent color. This method will always return a screen, even if no color screen is present.

See also: **NXBPSFromDepth()**

context

– (DPSContext)**context**

Returns the Application object's Display PostScript context.

currentEvent

– (NXEvent *)**currentEvent**

Returns a pointer to the last event the Application object retrieved from the event queue. A pointer to the current event is also passed with every event message.

See also: – **getNextEvent:waitFor:threshold:**, – **peekNextEvent:waitFor:threshold:**

deactivateSelf

– **deactivateSelf**

Deactivates the application if it's active. Normally, you shouldn't invoke this method; the Application Kit is responsible for proper deactivation. Returns **self**.

See also: – **activeApp**, – **activate:**, – **activateSelf:**

delayedFree:

– **delayedFree:***theObject*

Frees *theObject* by sending it the **free** message after the application finishes responding to the current event and before it gets the next event. If this method is performed during a modal loop, *theObject* is freed after the modal loop ends. Returns **self**.

See also: – **perform:with:afterDelay:cancelPrevious:** (DelayedPerform informal protocol)

delegate

– **delegate**

Returns the Application object's delegate.

See also: – **setDelegate:**

doesImportAlpha

– (BOOL)**doesImportAlpha**

Reports whether the application imports colors that include a value for alpha (opacity), and includes an opacity slider in its ColorPanel. The default is YES.

See also: – **setImportAlpha:**

endModalSession:

– **endModalSession:**(NXModalSession *)*session*

Cleans up after a modal session. The argument *session* should be from a previous invocation of **beginModalSession:for:**.

See also: – **runModalSession:**, – **beginModalSession:for:**

fileOperationCompleted:

– (int)**fileOperationCompleted:**(int)*operation*

Notification from the Workspace Manager that the file operation identified by *operation* has completed. The argument is the integer returned by the method that requested the file operation, to wit **performFileOperation:source:destination:files:options:** (part of NXWorkspaceRequest protocol).

If the delegate implements the method **app:fileOperationCompleted:**, that message is sent to it. If the delegate doesn't implement it, the method is handled by the Application subclass object (if you created one). The return is an arbitrary integer; your application defines and interprets it. If you neither provide a delegate method nor override in a subclass, the default definition simply returns 0.

findWindow:

– **findWindow:**(int)*windowNum*

Returns the Window object that corresponds to the window number *windowNum*. This method is of primary use in finding the Window object associated with a particular event.

See also: – **windowNum** (Window)

focusView

– **focusView**

Returns the View whose focus is currently locked, or **nil** if no View's focus is locked.

See also: – **lockFocus** (View)

free

– **free**

Closes all the Application object's windows, breaks the connection to the Window Server, and frees the Application object.

getNextEvent:

– (NXEvent *)**getNextEvent:**(int)*mask*

Gets the next event from the Window Server and returns a pointer to its event record. This method is similar to **getNextEvent:waitFor:threshold:** with an infinite timeout and a threshold of NX_MODALRESPTHRESHOLD.

See also: – **getNextEvent:waitFor:threshold**, – **run**, – **runModalFor:**, – **currentEvent**

getNextEvent:waitFor:threshold:

– (NXEvent *)**getNextEvent:**(int)*mask*
waitFor:(double)*timeout*
threshold:(int)*level*

Gets the next event from the Window Server and returns a pointer to its event record. Only events that match *mask* are returned; **getNextEvent:waitFor:threshold:** goes through the event queue, starting from the head, until it finds an event matching *mask*. Events that are skipped are left in the queue. Note that **getNextEvent:waitFor:threshold:** doesn't alter the window event masks that determine which events the Window Server will send to the application.

If an event matching the mask doesn't arrive within *timeout* seconds, this method returns a NULL pointer.

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

level determines what other procedures should be performed when the event queue is examined. These might include procedures to deal with timed-entries, procedures to handle messages received on ports, or procedures to read new data from files. Any such procedure that needs to be called will be called if its priority (specified when the procedure is registered) is equal to or higher than *level*.

In general, modal responders should pass NX_MODALRESPTHRESHOLD for *level*. The main run loop uses a threshold of NX_BASETHRESHOLD, allowing all procedures (except those registered with priority 0) to be checked and invoked if needed.

See also: – **peekNextEvent:waitFor:threshold:**, – **run**, – **runModalFor:**

getScreens:count:

– **getScreens:**(const NXScreen **)*list* **count:**(int *)*numScreens*

Gets screen information for every screen connected to the system. A pointer to an array of NXScreen structures is placed in the variable indicated by *list*, and the number of NXScreen structures in that array is placed in the variable indicated by *numScreens*. The list of NXScreen structures belongs to the Application object; it should not be altered or freed. Returns **self**.

getScreenSize:

– **getScreenSize:**(NXSize *)*theSize*

Gets the size of the main screen, in units of the screen coordinate system, and places it in the structure pointed to by *theSize*. Returns **self**.

getWindowNumbers:count:

– **getWindowNumbers:**(int **)*list* **count:**(int *)*numWindows*

Gets the window numbers for all the Application object's Windows. A pointer to a non-NULL-terminated array of **int**s is placed in the variable indicated by *list*. The number of entries in this array is placed in the integer indicated by *numWindows*. The order of window numbers in the array is the same as their order in the Window Server's screen list, which is their front-to-back order on the screen. The application is responsible for freeing the *list* array when done. Returns **self**.

See also: NXWindowList()

hide:

– **hide:***sender*

Collapses the application's graphics—including all its windows, menus, and panels—into a single small window. The **hide:** message is usually sent using the Hide command in the application's main Menu. Returns **self**.

See also: – unhide:

hostName

– (const char *)**hostName**

Returns the name of the host machine on which the Window Server that serves the Application object is running. This method returns the name that was passed to the receiving Application object through the NXHost default; this name is set either from its value in the defaults database or by providing a value for NXHost through the command line. If a value for NXHost isn't specified, NULL is returned.

isActive

– (BOOL)**isActive**

Returns YES if the application is currently active, and NO if it isn't.

See also: – **activateSelf:**, – **activate:**

isHidden

– (BOOL)**isHidden**

Returns YES if the application is currently hidden, and NO if it isn't.

isJournalable

– (BOOL)**isJournalable**

Returns YES if the application can be journaled, and NO if it can't. By default, applications can be journaled. Journaling is handled by the NXJournaler class.

See also: – **setJournalable:**

isRunning

– (BOOL)**isRunning**

Returns YES if the application is running, and NO if the **stop:** method has ended the main event loop.

See also: – **run**, – **stop:**, – **terminate:**

keyWindow

– **keyWindow**

Returns the key Window, that is, the Window that receives keyboard events. If there is no key Window, or if the key Window belongs to another application, this method returns **nil**.

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

loadNibFile:owner:

– **loadNibFile:**(const char *)*filename* **owner:***anOwner*

Loads interface objects from a NeXT Interface Builder (nib) file. The argument *anOwner* is the object that appears as the "File's Owner" in Interface Builder's File window. The objects and their names are read from the specified nib file into storage allocated from the default zone.

Objects that were *archived* in the nib file (standard objects from an Interface Builder palette) are sent **finishUnarchiving** and **awake** messages; other objects are *instantiated* and are sent an **init** message.

Returns non-**nil** if the file *filename* is successfully opened and read, and **nil** otherwise.

Invoking **loadNibFile:owner:** is equivalent to invoking **loadNibFile:owner:withNames:fromZone:** when the additional argument values indicate that names should also be loaded and that memory should be allocated from the default zone.

See also: – **loadNibFile:owner:withNames:fromZone:**, **NXDefaultMallocZone**(), – **awake** (Object), – **init** (Object)

loadNibFile:owner:withNames:

– **loadNibFile:**(const char *)*filename*
owner:*anObject*
withNames:(BOOL)*flag*

Loads interface objects from a NeXT Interface Builder (nib) file. The argument *anOwner* is the object that appears as the "File's Owner" in Interface Builder's File window. The objects are read from the specified interface file into storage allocated from the default zone. When *flag* is YES, the objects' names are also loaded. Names *must* be loaded if you use **NXGetNamedObject()** to get at the objects, but are not otherwise required.

Objects that were *archived* in the nib file (standard objects from an Interface Builder palette) are sent **finishUnarchiving** and **awake** messages; other objects are *instantiated* and are sent an **init** message.

Returns non-**nil** if the file *filename* is successfully opened and read.

Invoking **loadNibFile:owner:withNames:** is equivalent to invoking **loadNibFile:owner:withNames:fromZone:** when *zone* specifies that memory should be allocated from the default zone.

See also: – **loadNibFile:owner:withNames:fromZone:**, **NXDefaultMallocZone**(), – **awake** (Object), – **init** (Object)

loadNibFile:owner:withNames:fromZone:

– **loadNibFile:**(const char *)*filename*
owner:*anOwner*
withNames:(BOOL)*flag*
fromZone:(NXZone *)*zone*

Loads interface objects from a NeXT Interface Builder (nib) file. The argument *anOwner* is the object that appears as the "File's Owner" in Interface Builder's File window. The objects are read into memory allocated from *zone*. When *flag* is YES, the objects' names are also loaded. Names *must* be loaded if you use **NXGetNamedObject**() to get at the objects, but are not otherwise required. Objects that were *archived* in the nib file (standard objects from an Interface Builder palette) are sent **finishUnarchiving** and **awake** messages; other objects are *instantiated* and are sent an **init** message.

Returns non-**nil** if the file *filename* is successfully opened and read.

See also: – **awake** (Object), – **init** (Object)

loadNibSection:owner:

– **loadNibSection:**(const char *)*name* **owner:***anOwner*

Loads interface objects and their names from the source identified by *name*. To find the source, the method searches as follows:

- First, for a section named *name* within the __NIB segment of the application's executable file. (This is where earlier versions of Interface Builder routinely put nib sections, but not where Project Builder puts them now, so the section will be here only if the applications was compiled by an earlier version of Interface Builder.)

- Second, if no such section exists, the method searches certain language directories within the main bundle for a file with name *name* and type "nib," and—if it finds one—loads the interface objects from there. It searches the language directories that the user specified for this application, or (if none) those specified by the user's default language preferences (see **systemLanguages**).
- Third, if there's no file named *name* in the main bundle's relevant language directories, it looks for a file with name *name* and type "nib" in the main bundle (but outside the ".lproj" directories).

The argument *anOwner* is the object that corresponds to the "File's Owner" object in Interface Builder's File window. The loaded objects are allocated memory from the default zone.

Objects that were *archived* in the nib file (standard objects from an Interface Builder palette) are sent **finishUnarchiving** and **awake** messages; other objects are *instantiated* and are sent an **init** message.

Returns non-**nil** if the section or file is successfully opened and read.

Invoking **loadNibSection:owner:** is equivalent to invoking **loadNibSection:owner:withNames:fromZone:** when the additional arguments indicate that names should also be loaded and that memory should be allocated from the default zone.

See also: – **NXDefaultMallocZone**(), + **mainBundle** (NXBundle), – **getPath:forResource:ofType:** (NXBundle), – **awake** (Object), – **init** (Object)

loadNibSection:owner:withNames:

– **loadNibSection:**(const char *)*name*
owner:*anOwner*
withNames:(BOOL)*flag*

Loads interface objects and their names from the source identified by *name*. The source may be a section within the executable file, or a file within the application bundle, as described above for the **loadNibSection:owner:** instance method.

The argument *anOwner* is the object that corresponds to the "File's Owner" object in Interface Builder's File window. The loaded objects are allocated memory from the default zone. When *flag* is YES, the objects' names are also loaded. Names *must* be loaded if you use **NXGetNamedObject**() to get at the objects, but are not otherwise required.

Objects that were *archived* in the nib file (standard objects from an Interface Builder palette) are sent **finishUnarchiving** and **awake** messages; other objects are *instantiated* and are sent an **init** message.

Returns non-**nil** if the section or file is successfully opened and read.

Invoking **loadNibSection:owner:withNames** is equivalent to invoking **loadNibSection:owner:withNames:fromZone:** when the additional argument indicates that memory should be allocated from the default zone.

See also: **NXDefaultMallocZone**(), – **awake** (Object), – **init** (Object)

loadNibSection:owner:withNames:fromHeader:

– **loadNibSection:**(const char *)*name*
owner:*anOwner*
withNames:(BOOL)*flag*
fromHeader:(const struct mach_header *)*header*

Loads interface objects from a section within a dynamically loaded object file—that is, from a file other than those in the application's main bundle. The argument *header* identifies the file, as returned by the function **objc_loadModule()**. The argument *name* identifies a named section within the file's __NIB segment. When no such file exists, the method searches the executable file's bundle, first within its language subdirectories, as described above for the **loadNibSection:owner:** instance method.

The argument *anOwner* is the object that corresponds to the "File's Owner" object in Interface Builder's File window. Memory for the loaded objects is allocated from the default zone. When *flag* is YES, the objects' names are also loaded. Names *must* be loaded if you use **NXGetNamedObject()** to get at the objects, but are not otherwise required.

Objects that were *archived* in the nib file (standard objects from an Interface Builder palette) are sent **finishUnarchiving** and **awake** messages; other objects are *instantiated* and are sent an **init** message.

A class can use this method in its **finishLoading** class method to load interface data objects required by the class but stored separately (for example, because the same interface objects are also used by other classes).

Returns non-**nil** if the section or file is successfully opened and read.

Invoking **loadNibSection:owner:withNames:fromHeader:** is equivalent to invoking **loadNibSection:owner:withNames:fromHeader:fromZone:** when the additional arguments indicate that names should also be loaded and that memory should be allocated from the default zone.

See also: **NXDefaultMallocZone**(), – **awake** (Object), – **init** (Object)

loadNibSection:owner:withNames:fromHeader:fromZone:

– **loadNibSection:**(const char *)*name*
owner:*anOwner*
withNames:(BOOL)*flag*
fromHeader:(const struct mach_header *)*header*
fromZone:(NXZone *)*zone*

Loads interface objects from a section within a dynamically loaded object file—that is, from a file other than those in the application's main bundle. The argument *header* identifies the file, as returned by the function **objc_loadModule()**. The argument *name* identifies a named section within the file's __NIB segment. When no such file exists, the method searches the executable file's bundle, first within its language subdirectories, as described above for the **loadNibSection:owner:** instance method.

The argument *anOwner* is the object that corresponds to the "File's Owner" object in Interface Builder's File window. Memory for the loaded objects is allocated from the zone specified by *zone*. When *flag* is YES, the objects' names are also loaded. Names *must* be loaded if you use **NXGetNamedObject()** to get at the objects, but are not otherwise required. Objects that were *archived* in the nib file (standard objects from an Interface Builder palette) are sent **finishUnarchiving** and **awake** messages; other objects are *instantiated* and are sent an **init** message.

A class can use this method in its **finishLoading** class method to load interface data objects required by the class but stored separately (for example, because the same interface objects are also used by other classes).

Returns non-**nil** if the section is successfully opened and read.

See also: – **loadNibSection:owner:withNames:fromZone:**, – **awake** (Object), – **init** (Object)

loadNibSection:owner:withNames:fromZone:

– **loadNibSection:**(const char *)*name*
owner:*anOwner*
withNames:(BOOL)*flag*
fromZone:(NXZone *)*zone*

Loads interface objects and their names from the source identified by *name*. The source may be a section within the executable file, or a file within the application bundle, as described above for the **loadNibSection:owner:** instance method.

The argument *anOwner* is the object that corresponds to the "File's Owner" object in Interface Builder's File window. When *flag* is YES, the objects' names are also loaded. Names *must* be loaded if you use **NXGetNamedObject()** to get at the objects, but are not otherwise required. Memory for the loaded objects is allocated from the zone specified by *zone*. Objects that were *archived* in the nib file (standard objects from an Interface Builder palette) are sent **finishUnarchiving** and **awake** messages; other objects are *instantiated* and are sent an **init** message.

Returns non-**nil** if the section or file is successfully opened and read, and **nil** otherwise.

See also: – **loadNibSection:owner:withNames:fromHeader:fromZone:**, – **awake** (Object), – **init** (Object)

mainMenu

– **mainMenu**

Returns the Application object's main Menu.

mainScreen

– (const NXScreen *)**mainScreen**

Returns the main screen. If there is only one screen, that screen is returned. Otherwise, this method attempts to return the key window's screen. If there is no key window, it attempts to return the main menu's screen. If there is no main menu, this method returns the screen that contains the screen coordinate system origin.

See also: – **screen** (Window)

mainWindow

– **mainWindow**

Returns the main Window. This method returns **nil** if there is no main window, if the main window belongs to another application, or if the application is hidden.

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

makeWindowsPerform:inOrder:

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

Sends the Application object's Windows a message to perform the *aSelector* method. The message is sent to each Window in turn until one of them returns YES; this method then returns that Window. If no Window returns YES, this method returns **nil**.

If *flag* is YES, the Application object's Windows receive the *aSelector* message in the front-to-back order in which they appear in the Window Server's window list. If *flag* is NO, Windows receive the message in the order they appear in the Application object's window list. This order generally reflects the order in which the Windows were created.

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

masterJournaler

– **masterJournaler**

Returns the Application object's master journaler. Journaling is handled by the NXJournaler class.

See also: – **slaveJournalar:**

miniaturizeAll:

– **miniaturizeAll:***sender*

This method miniaturizes all of the receiver's application windows. Returns **self**.

mounted:

– (int)**mounted:**(const char *)*fullPath*

Invoked by the Workspace Manager when the device identified by *fullPath* has completed mounting. You shouldn't directly send a **mounted:** message. This is one of the messages the Application will receive if it has previously sent the Workspace Manager the message **beginListeningForDeviceStatusChanges**.

If the delegate implements the method **app:mounted:**, that message is sent to it. If the delegate doesn't implement it, the method is handled by the Application subclass object (if you created one). The return value is an arbitrary integer; your application defines and interprets it. If you neither provide a delegate method nor override in a subclass, the default definition simply returns 0.

See also: – **unmounting:ok:**, – **unmounted:**

openFile:ok:

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

Responds to a remote message requesting the application to open a file. **openFile:ok:** is typically sent to the application from the Workspace Manager, although an application can send it directly to another application. The Application object's delegate is queried with **appAcceptsAnotherFile:** and if the result is YES, it's sent an **app:openFile:type:** message. If the delegate doesn't respond to either of these messages, they're sent to the Application object (if it implements them).

The variable pointed to by *flag* is set to YES if the file is successfully opened, NO if the file is not successfully opened, and –1 if the application does not accept another file. Returns zero.

See also: – **app:openFile:type:** (delegate method), – **openTempFile:ok:**, – **openFile:ok:** (Speaker)

openTempFile:ok:

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

Same as the **openFile:ok:** method, but **app:openTempFile:type:** is sent. Returns 0.

See also: – **app:openTempFile:type:** (delegate method), – **openTempFile:ok:** (Speaker)

orderFrontColorPanel:

– **orderFrontColorPanel:***sender*

Displays the color panel. Returns **self**.

orderFrontDataLinkPanel:

#import NXDataLinkPanel.h
– **orderFrontDataLinkPanel:***sender*

Displays the data link panel. It does this by sending an **orderFront**: message to the shared instance of NXDataLinkPanel (if need be, creating a new one). Returns **self**.

peekAndGetNextEvent:

– (NXEvent *)**peekAndGetNextEvent:**(int)*mask*

This method is similar to **getNextEvent:waitFor:threshold:** with a zero timeout and a threshold of NX_MODALRESPTHRESHOLD.

See also: – **getNextEvent:waitFor:threshold**, – **run**, – **runModalFor:**, – **currentEvent**, – **peekNextEvent:into:**

peekNextEvent:into:

– (NXEvent *)**peekNextEvent:**(int)*mask* **into:**(NXEvent *)*eventPtr*

This method is similar to **peekNextEvent:into:waitFor:threshold:** with a zero timeout and a threshold of NX_MODALRESPTHRESHOLD.

See also: – **peekNextEvent:into:waitFor:threshold**, – **run**, – **runModalFor:**, – **currentEvent**

peekNextEvent:into:waitFor:threshold:

– (NXEvent *)**peekNextEvent:**(int)*mask*
into:(NXEvent *)*eventPtr*
waitFor:(float)*timeout*
threshold:(int)*level*

This method is similar to **getNextEvent:waitFor:threshold:** except the matching event isn't removed from the event queue nor is it placed in **currentEvent**; instead, it's copied into storage pointed to by *eventPtr*.

If no matching event is found, NULL is returned; otherwise, *eventPtr* is returned.

See also: – **getNextEvent:waitFor:threshold:**, – **run**, – **runModalFor:**, – **currentEvent**

powerOff:

– **powerOff:**(NXEvent *)*theEvent*

A **powerOff:** message is generated when a power-off event is sent from the Window Server. As a general rule, only the Workspace Manager and login window should respond to this event. If the application was launched by the Workspace Manager, this method does nothing; instead, the Application object will wait for the **powerOffIn:andSave:** message from the Workspace Manager. If the application wasn't launched from the Workspace

Manager, this method sends the delegate a **powerOff:** message, assuming there's a delegate and it implements the method. Applications that are not launched from the Workspace Manager are not fully supported, and are not guaranteed any amount of time after receiving this message. However, applications launched from the Workspace Manager can request additional time before shutdown from within the **app:powerOffIn:andSave** method. Returns **self**.

See also: – **app:powerOffIn:andSave:** (delegate method), – **powerOffIn:andSave:**

powerOffIn:andSave:

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

You never invoke this method directly; it's sent from the Workspace Manager. The delegate or your subclass of Application will be given the chance to receive the **app:powerOffIn:andSave** message. The *aFlag* parameter has no particular meaning and can be ignored. This method raises an exception, so it never returns.

See also: – **app:powerOffIn:andSave:** (delegate method)

preventWindowOrdering

– **preventWindowOrdering**

Suppresses the usual window ordering behavior entirely. Most applications will not need to use this method since the Application Kit support for dragging will call it when dragging is initiated.

printInfo

– **printInfo**

Returns the Application object's global PrintInfo object. If none exists, a default one is created.

registerServicesMenuSendTypes:andReturnTypes:

– **registerServicesMenuSendTypes:**(const char *const *)*sendTypes*
andReturnTypes:(const char *const *)*returnTypes*

Registers pasteboard types that the application can send and receive in response to service requests. If the application has a Services menu, a menu item is added for each service provider that can accept one of the specified send types or return one of the specified return types. This method should typically be invoked at application startup time or when an object that can use services is created. It can be invoked more than once; its purpose is to ensure that there is a menu item for every service that the application may use. The individual items will be dynamically enabled and disabled by the event handling mechanism to indicate which services are currently appropriate. An application (or object instance that can cut or paste) should register every possible type that it can send and receive. Returns **self**.

See also: – **validRequestorForSendType:andReturnType:** (Responder),
– **readSelectionFromPasteboard:** (Object method),
– **writeSelectionToPasteboard:types:** (Object method)

removeWindowsItem:

– **removeWindowsItem:***aWindow*

Removes the item for *aWindow* in the Windows menu. Note that this method doesn't prevent the item from being automatically added again, so you must use Window's **setExcludedFromWindowsMenu:** method if you want the item to remain excluded from the Windows menu. Returns **self**.

See also: – **changeWindowsItem:title:filename:**,
– **setExcludedFromWindowsMenu:** (Window)

replyPort

– (port_t)**replyPort**

Returns the Application object's reply port. This port is allocated for you automatically by the **run** method, and is the default reply port which can be shared by all the Application object's Speakers.

See also: – **setReplyPort:** (Speaker)

resignActiveApp

– **resignActiveApp**

This method is invoked immediately after the application is deactivated. You never send **resignActiveApp** messages directly, but you could override this method in your Application object to notice when your application is deactivated. Alternatively, your delegate could implement **appDidResignActive:**. Returns **self**.

See also: – **deactivateSelf:**, – **appDidResignActive:** (delegate method)

rightMouseDown:

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

Pops up the main Menu. Returns **self**.

run

– **run**

Initiates the Application object's main event loop. The loop continues until a **stop:** or **terminate:** message is received. Each iteration through the loop, the next available event from the Window Server is stored, and is then dispatched by sending the event to the Application object using **sendEvent:**

A **run** message should be sent as the last statement from **main()**, after the application's objects have been initialized. Returns **self** if terminated by **stop:**, but never returns if terminated by **terminate:**.

See also: – **runModalFor:**, – **sendEvent:**, – **stop:**, – **terminate:**, – **appDidInit:** (delegate method)

runModalFor:

– (int)**runModalFor:***theWindow*

Establishes a modal event loop for *theWindow*. Until the loop is broken by a **stopModal**, **stopModal:**, or **abortModal** message, the application won't respond to any mouse, keyboard, or window-close events unless they're associated with *theWindow*. If

stopModal: is used to stop the modal event loop, this method returns the argument passed to **stopModal:**. If **stopModal** is used, it returns the constant NX_RUNSTOPPED. If **abortModal** is used, it returns the constant NX_RUNABORTED. This method is functionally similar to the following:

```
NXModalSession session;
[NXApp beginModalSession:&session for:theWindow];
for (;;) {
   if ([NXApp runModalSession:&session] != NX_RUNCONTINUES)
        break;
}
[NXApp endModalSession:&session];
```

See also: – **stopModal**, – **stopModal:**, – **abortModal**, – **runModalSession:**

runModalSession:

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

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

If the modal session was not stopped, this method returns NX_RUNCONTINUES. If **stopModal** was invoked as the result of event procession, NX_RUNSTOPPED is returned. If **stopModal:** was invoked, this method returns the value passed to **stopModal:**. The NX_abortModal exception raised by **abortModal** isn't caught.

See also: – **beginModalSession:**, – **endModalSession**, – **stopModal:**, – **stopModal**, – **runModalFor:**

runPageLayout:

– **runPageLayout:***sender*

Brings up the Application object's Page Layout panel, which allows the user to select the page size and orientation. Returns **self**.

sendAction:to:from:

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

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

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

Returns YES if the action is applied; otherwise returns NO.

sendEvent:

– **sendEvent:**(NXEvent *)*theEvent*

Sends an event to the Application object. You rarely send **sendEvent:** messages directly although you might want to override this method to perform some action on every event. **sendEvent:** messages are sent from the main event loop (the **run** method). **sendEvent** is the method that dispatches events to the appropriate responders; the Application object handles application events, the Window indicated in the event record handles window related events, and mouse and key events are forwarded to the appropriate Window for further dispatching. Returns **self.**

See also: – **setAutoupdate:**

servicesMenu

– **servicesMenu**

Returns the Application object's Services menu. Returns **nil** if no Services menu has been created.

See also: – **setServicesMenu:**

setAppListener:

– **setAppListener:***aListener*

Sets the Listener that will receive messages sent to the port that's registered for the application. If you want to have a special Listener reply to these messages, you must either send a **setAppListener:** message before the **run** message is sent to the Application object, or send this message from the delegate method **appWillInit:**, so that *aListener* is properly registered. This method doesn't free the Application object's previous Listener object. Returns **self**.

See also: – **appListenerPortName**, – **appWillInit:** (delegate method)

setAppSpeaker:

– **setAppSpeaker:***aSpeaker*

Sets the Application object's Speaker. If you don't send a **setAppSpeaker:** message before the Application object initializes, a default Speaker is created for you. This method doesn't free the Application object's previous Speaker object.

See also: – **appWillInit:** (delegate method)

setAutoupdate:

– **setAutoupdate:**(BOOL)*flag*

Turns on or off automatic updating of the application's windows. (Until this message is sent, automatic updating is not enabled.) When automatic updating is on, an update message is sent to each of the application's Windows after each event has been processed. This can be used to keep the appearance of menus and panels synchronized with your application. Returns **self**.

See also: – **updateWindows**

setDelegate:

– **setDelegate:***anObject*

Sets the Application object's delegate. The notification messages that a delegate can expect to receive are listed at the end of the Application class specification. The delegate doesn't need to implement all the methods. Returns **self**.

See also: – **delegate**

setImportAlpha:

– **setImportAlpha:**(BOOL)*flag*

Determines whether your application will accept translucent colors in objects it receives. This affects colors imported by the View method **acceptsColor:atPoint:**, or by NXColorPanel's **dragColor:withEvent:fromView:**. It has no effect on internal programmatic manipulations of colors.

A pixel may be described by its color (values for red, blue, and green) and also by its opacity, measured by a coefficient called alpha. When alpha is 1.0, a color is completely opaque and thus hides anything beneath it. When alpha is less then 1, the effective color is derived partly from the color of the object itself and partly from the color of whatever is beneath it. When *flag* is YES, the application accepts a color that includes an alpha coefficient, and forces an alpha value of 1.0 for a source where alpha was not specified. In addition, when *flag* is YES, a ColorPanel opened within the application includes an opacity slider.

When the Application has received a **setImportAlpha:** message with *flag* set to NO, all imported colors are forced to have an alpha value of NX_NOALPHA, and there's no opacity slider in the ColorPanel. The default state is NO, do not import alpha.

This method has the same effect as the NXColorPanel method **setShowAlpha:**. The only difference is that you can invoke **setImportAlpha:** even before an NXColorPanel has been instantiated. Since the two methods set the same internal flag, each can reverse the effect of the other.

Returns **self**.

See also: – **doesImportAlpha**, – **doesShowAlpha** (NXColorPanel), – **setShowAlpha:** (NXColorPanel)

setJournalable:

– **setJournalable:**(BOOL)*flag*

Sets whether the application is journalable. Returns **self**. Journaling is handled by the NXJournaler class.

See also: – **isJournalable**

setMainMenu:

– **setMainMenu:***aMenu*

Makes *aMenu* the Application object's main menu. Returns **self**.

See also: – **mainMenu**

setPrintInfo:

– **setPrintInfo:***info*

Sets the Application object's global PrintInfo object. Returns the previous PrintInfo object, or **nil** if there was none.

See also: – **printInfo**

setServicesMenu:

– **setServicesMenu:***aMenu*

Makes *aMenu* the Application object's Services menu. Returns **self**.

See also: – **servicesMenu**

setWindowsMenu:

– **setWindowsMenu:***aMenu*

Makes *aMenu* the Application object's Windows menu. Returns **self**.

See also: – **windowsMenu**

showHelpPanel:

– **showHelpPanel:***sender*

Shows the application's Help panel. If no Help panel yet exists, the method first creates a default Help panel. If the delegate implements **app:willShowHelpPanel:**, notifies it. Returns **self**.

slaveJournaler

– **slaveJournaler**

Returns the Application object's slave journaler. Journaling is handled by the NXJournaler class.

See also: – **masterJournalar:**

stop:

– **stop:***sender*

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

If this method is applied during a modal event loop, it will break that loop but not the main event loop. Returns **self**.

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

stopModal

– **stopModal**

Stops a modal event loop. This method should always be paired with a previous **runModalFor:** or **beginModalSession:for:** message. When **runModalFor:** is stopped with this method, it returns NX_RUNSTOPPED. This method will stop the loop only if it's executed by code responding to an event. If you need to stop a **runModalFor:** loop from a procedure registered with **DPSAddTimedEntry()**, **DPSAddPort()**, or **DPSAddFD()**, use the **abortModal** method. Returns **self**.

See also: – **stopModal:**, – **runModalFor:**, – **runModalSession:**, – **abortModal**

stopModal:

– **stopModal:**(int)*returnCode*

Just like **stopModal** except argument *returnCode* allows you to specify the value that **runModalFor:** will return. Returns **self**.

See also: – **stopModal**, – **runModalFor:**, – **abortModal**

systemLanguages

– (const char *const *)**systemLanguages**

Returns a list of the names of languages in order of the user's preference. If your application will respond to the user's language preference, this method is the way to discover what the preferences are. The return is a NULL-terminated list of pointers to NULL-terminated strings.

If the user has recorded preferences specific to the application now in use, the method returns them. If the user has recorded no preferences for the application, but has recorded a global preference, the method returns the list of global preferences. (Note that just because the user has recorded a preference doesn't mean than the language files are in fact installed on the host that is executing the application.) If this method returns NULL, the user has no language preference.

terminate:

– **terminate:***sender*

Terminates the application. (This is the default action method for the application's Quit menu item.) Each use of **terminate:** invokes **appWillTerminate:** to notify the delegate that the application will terminate. If **appWillTerminate:** returns **nil**, **terminate:** returns **self**; control is returned to the main event loop, and the application isn't terminated. Otherwise, this method frees the Application object and calls **exit()** to terminate the application. Note that you should not put final cleanup code in your application's **main()** function; it will never be executed.

See also: – **stop**, – **appWillTerminate:** (delegate method), **exit()**

tryToPerform:with:

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

Aids in dispatching action messages. The Application object tries to perform the method *aSelector* using its inherited Responder method **tryToPerform:with:**. If the Application object doesn't perform *aSelector*, the delegate is given the opportunity to perform it using its inherited Object method **perform:with:**. If either the Application object or the Application object's delegate accept *aSelector*, this method returns YES; otherwise it returns NO.

See also: – **tryToPerform:with:** (Responder), – **respondsTo:** (Object), – **perform:with:** (Object)

unhide

– (int)**unhide**

Responds to an **unhide** message sent from Workspace Manager. You shouldn't invoke this method; invoke **unhide:** instead. Returns zero.

See also: – **unhide:**

unhide:

– **unhide:***sender*

Restores a hidden application to its former state (all of the windows, menus, and panels visible), and makes it the active application. This method is usually invoked as the result of double-clicking the icon for the hidden application. Returns **self**.

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

unhideWithoutActivation:

– **unhideWithoutActivation:***sender*

Unhides the application but doesn't make it the active application. You might want to invoke **activateSelf:**NO after invoking this method to make the receiving application active if there is no active application. Returns **self**.

See also: – **hide:**, – **activateSelf:**

unmounted:

– (int)**unmounted:**(const char *)*fullPath*

Invoked by the Workspace Manager when it has completed unmounting the device identified by *fullPath*. You shouldn't directly send an **unmounted:** message. This is one of the messages the Application will receive if it has previously sent the Workspace Manager the message **beginListeningForDeviceStatusChanges**.

If the delegate implements the method **app:unmounted:**, that message is sent to it. If the delegate doesn't implement it, the method is handled by the Application subclass object (if you created one). The return is an arbitrary integer; your application defines and interprets it. If you neither provide a delegate method nor override in a subclass, the default definition simply returns 0.

See also: – **mounted:**, – **unmounting:ok:**

unmounting:ok:

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

Invoked and sent to all active applications when the Workspace Manager has received a request to unmount the device identified by *fullPath*. This serves to warn applications that may be making use of the device. You shouldn't directly send **unmounting:ok:** messages.

The method sets *flag* to point to YES to indicate that the Application assents to unmounting, and NO if it objects.

If the delegate implements the method **app:unmounting:**, that message is sent to it, and *flag* is set to whatever the delegate returns. If the delegate doesn't implement **app:unmounting:**, the method is handled by the Application subclass object (if you created one). The default behavior is to close all files on the device, and if the current working directory is on the device, to change the current working directory to the user's home directory.

The return value is an arbitrary integer; your application defines and interprets it. If you neither provide a delegate method nor override in a subclass, the default definition simply returns 0.

updateWindows

– **updateWindows**

Sends an **update** message to the Application object's visible Windows. When automatic updating has been enabled, this method is invoked automatically in the main event loop after each event. An application can also send **updateWindows** messages at other times to have Windows update themselves.

If the delegate implements **appWillUpdate:**, that message is sent to the delegate before the windows are updated. Similarly, if the delegate implements **appDidUpdate:**, that message is sent to the delegate after the windows are updated. Returns **self**.

See also: – **setAutoupdate:**, – **appWillUpdate:** (delegate method), – **appDidUpdate:** (delegate method)

updateWindowsItem:

– **updateWindowsItem:***aWindow*

Updates the item for *aWindow* in the Windows menu to reflect the edited status of *aWindow*. You rarely need to invoke this method because it is invoked automatically when the edited status of a Window is set. Returns **self**.

See also: – **changeWindowsItem:title:filename:**, – **setDocEdited:** (Window)

validRequestorForSendType:andReturnType:

– **validRequestorForSendType:**(NXAtom)*sendType*
andReturnType:(NXAtom)*returnType*

Passes this message on to the Application object'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 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.

See also: – **validRequestorForSendType:andReturnType:** (Responder),
– **registerServicesMenuSendTypes:andReturnTypes:**,
– **writeSelectionToPasteboard:types:** (Object),
– **readSelectionFromPasteboard:** (Object)

windowList

– **windowList**

Returns the List object used to keep track of all the Application object's Windows, including Menus, Panels, and the like. In the current implementation, this list also contains global (shared) Windows.

windowsMenu

– **windowsMenu**

Returns the Application object's Windows menu. Returns **nil** if no Windows menu has been created.

Methods Implemented by the Delegate

app:applicationDidLaunch:

– **app:***sender* **applicationDidLaunch:**(const char *)*appName*

Implement this method to respond to an **applicationDidLaunch**: message sent from the Workspace Manager to *sender* (an Application object), informing it that an application named *appName* has launched. This is one of the messages the Application will receive if it has previously sent the Workspace Manager the message **beginListeningForApplicationStatusChanges**.

See also: – **applicationDidLaunch**:

app:applicationDidTerminate:

– **app:***sender* **applicationDidTerminate:**(const char *)*appName*

Implement this method to respond to an **applicationDidTerminate**: message sent from the Workspace Manager to *sender* (an Application object), informing it that an application named *appName* has terminated. This is one of the messages the Application will receive if it has previously sent the Workspace Manager the message **beginListeningForApplicationStatusChanges**.

See also: – **applicationDidTerminate**:

app:applicationWillLaunch:

– **app:***sender* **applicationWillLaunch:**(const char *)*appName*

Implement this method to respond to an **applicationWillLaunch**: message sent from the Workspace Manager to *sender* (an Application object), informing it that an application named *appName* is about to launch. This is one of the messages the Application will receive if it has previously sent the Workspace Manager the message **beginListeningForApplicationStatusChanges**.

See also: – **applicationWillLaunch:**

app:fileOperationCompleted:

– **app:***sender* **fileOperationCompleted:**(int)*operation*

Sent to the delegate when *sender* (an application) has completed the file operation identified by *operation*. The argument is the integer returned by the method that requested the file operation: **performFileOperation:source:destination:files:options:** (part of NXWorkspaceRequest protocol).

app:mounted:

– **app:***sender* **mounted:**(const char *)*fullPath*

Implement this method to respond to a **mounted**: message sent from the Workspace Manager to *sender* (an Application object), informing it that a device (for example a floppy disk or an optical disk) has been mounted. This is one of the messages the Application will receive if it has previously sent the Workspace Manager the message **beginListeningForDeviceStatusChanges**.

See also: – **mounted:**

app:openFile:type:

– (int)**app:***sender*
openFile:(const char *)*filename*
type:(const char *)*aType*

Invoked from within **openFile:ok:** after it has been determined that the application can open another file. The method should attempt to open the file of type *type* and name *filename*, returning YES if the file is successfully opened, and NO otherwise. (Although a file's type may by convention be reflected in its name, *type* is not a synonym for extension. *filename* should not exclude part of the name just because it can sometimes be inferred from type.)

This method is also invoked from within **openTempFile:ok:** if neither the delegate nor the Application subclass responds to **app:openTempFile:type:**

See also: – **openFile:ok:**, – **openTempFile:ok:**, –**app:openFileWithoutUI:type:**, – **app:openTempFile:type:**

app:openFileWithoutUI:type:

– (NXDataLinkManager *)**app:***sender*
openFileWithoutUI:(const char *)*filename*
type:(const char *)*type*

Sent to the delegate when *sender* (an Application) requests that the file of type *type* and name *filename* be opened as a linked file. The file is to be opened without bringing up its application's user interface; that is, work with the file will be under programmatic control of *sender*, rather than under keyboard control of the user.

Returns a pointer to the NXDataLinkManager that will coordinate data flow between the two applications.

See also: – **app:openFile:type:**

app:openTempFile:type:

– (int)**app:***sender*
openTempFile:(const char *)*filename*
type:(const char *)*aType*

Invoked from within **openTempFile:ok:** after it has been determined that the application can open another file. The method should attempt to open the file *filename* with the extension *aType*, returning YES if the file is successfully opened, and NO otherwise.

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

See also: – **openFile:ok:**, – **openTempFile:ok:**

app:powerOffIn:andSave:

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

Invoked from the **powerOffIn:andSave:** method after the Workspace Manager receives a power-off event. This method is invoked only if the application was launched from the Workspace Manager. The argument *ms* is the number of milliseconds to wait before powering down or logging out. The argument *aFlag* has no particular meaning at this time, and can be ignored. You can ask for additional time by sending the **extendPowerOffBy:actual:** message to the Workspace Manager from within your implementation of this method. The Workspace Manager will power the machine down (or log out the user) as soon as all applications terminate, even if there's time remaining on the time extension.

See also: – **extendPowerOffBy:actual:** (Speaker)

app:unmounted:

– **app:***sender* **unmounted:**(const char *)*fullPath*

Implement this method to respond to an **unmounted**: message sent from the Workspace Manager to *sender* (an Application object), informing it that the device identified by *fullPath* has been unmounted. This is one of the messages the Application will receive if it has previously sent the Workspace Manager a **beginListeningForDeviceStatusChanges** message.

See also: – **unmounted**, – **app:mounted:**

app:unmounting:

– (int)**app:***sender* **unmounting:**(const char *)*fullPath*

Invoked when the device mounted at *fullPath* is about to be unmounted. This method is invoked from **unmounting:ok:** and is invoked only if the application was launched from the Workspace Manager. The Application object or its delegate should do whatever is necessary to allow the device to be unmounted. Specifically, all files on the device should be closed and the current working directory should be changed if it's on the device.

See also: – **unmounting:ok:**, – **app:unmounted:**

app:willShowHelpPanel:

– **app:***sender* **willShowHelpPanel:***panel*

Implement this to respond to notice that *sender* (an Application) has received a **showHelpPanel:** message and is about to put up the Help panel identified by *panel*. The return value doesn't matter.

See also: – **showHelpPanel:**

appAcceptsAnotherFile:

– (BOOL)**appAcceptsAnotherFile:***sender*

Invoked from within Application's **openFile:ok:** and **openTempFile:ok:** methods, this method should return YES if it's okay for the application to open another file, and NO if isn't. If neither the delegate nor the Application object responds to the message, then the file shouldn't be opened.

See also: – **openFile:ok:**, – **openTempFile:ok:**

appDidBecomeActive:

– **appDidBecomeActive:***sender*

Implement to respond to notification sent from the Workspace Manager immediately after the Application becomes active.

See also: – **applicationDidLaunch:**

appDidHide:

– **appDidHide:***sender*

Invoked immediately after the application is hidden.

See also: – **hide**:, – **unhide**:, – **appDidUnhide:** (delegate method)

appDidInit:

– **appDidInit:***sender*

Invoked after the application has been launched and initialized, but before it has received its first event. The delegate or the Application subclass can implement this method to perform further initialization.

See also: – **appWillInit:** (delegate method)

appDidResignActive:

– **appDidResignActive:***sender*

Invoked immediately after the application is deactivated.

See also: – **becomeActiveApp**, – **resignActiveApp**

appDidUnhide:

– **appDidUnhide:***sender*

Invoked immediately after the application is unhidden.

See also: – **hide**:, – **unhide**:, – **appDidHide:** (delegate method)

appDidUpdate:

– **appDidUpdate:***sender*

Invoked immediately after the Application object updates its Windows.

See also: – **updateWindows**, – **updateWindowsItem:**, – **appWillUpdate:** (delegate method)

applicationDefined:

– **applicationDefined:**(NXEvent *)*theEvent*

Invoked when the application receives an application-defined (NX_APPDEFINED) event. See the description of this method under "Instance Methods," above.

appWillInit:

– **appWillInit:***sender*

Invoked before the Application object is initialized. This method is invoked before the Application object has initialized its Listener and Speaker objects and before any **app:openFile:type:** messages are sent to your delegate. The Application object's Listener and Speaker objects will be created for you immediately after invoking this method if they have not been previously created.

See also: – **appDidInit:** (delegate method), – **appListener**, – **appSpeaker**

appWillTerminate:

– **appWillTerminate:***sender*

Invoked from within the **terminate:** method immediately before the application terminates. If this method returns **nil**, the application is not terminated, and control is returned to the main event loop. If you want to allow the application to terminate, you should put your clean up code in this method and return non-**nil**.

See also: – **terminate:**

appWillUpdate:

– **appWillUpdate:***sender*

Invoked immediately before the Application object updates its Windows.

See also: – **updateWindows**, – **updateWindowsItem:**, – **appDidUpdate:** (delegate method)

powerOff:

– **powerOff:**(NXEvent *)*theEvent*

Invoked from the **powerOff:** Application method only if the application *wasn't* launched from the Workspace Manager. Only applications launched from the Workspace Manager are fully supported, so your application isn't guaranteed any amount of processing time after this message is received. This notification is provided mainly for the use of alternate login window programs.

See also: – **powerOff:**, – **powerOffIn:andSave:**

Box

Inherits From: View : Responder : Object

Declared In: appkit/Box.h

Class Description

A Box object is a simple View that can do two things: It can draw a border around itself and it can title itself. You use a Box to group, visually, some number of other Views. These other Views are added to the Box through the typical subview-adding methods, such as **addSubview:** and **replaceSubview:with:**.

A Box contains a *content area*, a rectangle set within the Box's frame in which the Box's subviews are displayed. The size and location of the content area depends on the Box's border type, title location, the size of the font used to draw the title, and an additional measure that you can set through the **setOffsets::** method. When you create a Box, a instance of View is created and added (as a subview of the Box object) to fill the Box's content area. If you replace this *content view* with a View of your own, your View will be resized to fit the content area. Similarly, as you resize a Box its content view is automatically resized to fill the content area.

The Views that you add as subviews to a Box are actually added to the Box's content view—View's subview-adding methods are redefined by Box to ensure that a subview is correctly placed in the view hierarchy. However, you should note that the **subviews** method *isn't* redefined: It returns a List containing a single object, the Box's content view.

Instance Variables

id **cell**;
id **contentView**;
NXSize **offsets**;
NXRect **borderRect**;
NXRect **titleRect**;

```
struct _bFlags {
        unsigned int borderType:2;
        unsigned int titlePosition:3;
} bFlags;
```

cell	The cell that draws the Box's title.
contentView	The Views that fills the Box's content area.
offsets	Offsets of the content view from the Box's border.
borderRect	The Box's border rectangle.
titleRect	The rectangle in which the title cell is drawn.
bFlags.borderType	A constant describing the Box's border type.
bFlags.titlePosition	A constant describing the position of the Box's title.

Method Types

Initializing and freeing	– initFrame: – free
Setting the border and title	– setBorderType: – borderType – setTitlePosition: – titlePosition – setTitle: – title – setFont: – font – cell
Setting and placing the content view	– setContentView: – contentView – setOffsets:: – getOffsets:
Putting Views in the Box	– addSubview: – replaceSubview:with:
Resizing the Box	– setFrameFromContentFrame: – sizeTo:: – sizeToFit

Drawing the Box	– drawSelf::
Archiving	– awake – read: – write:

Instance Methods

addSubview:

– **addSubview:***aView*

Adds *aView* to the Box. This is done by forwarding the **addSubview:***aView* message to the Box's content view. Note that this means *aView*'s location and size are reckoned within the content view's coordinate system. After invoking this method, you should send the Box a **sizeToFit** message. Returns **self**.

awake

– **awake**

Lays out the Box during the unarchiving process so that it can be displayed. You should never invoke this method directly.

borderType

– (int)**borderType**

Returns the Box's border type, one of NX_LINE, NX_GROOVE, NX_BEZEL, or NX_NOBORDER. By default, a Box's border type is NX_GROOVE.

See also: – **setBorderType:**

cell

– **cell**

Returns the cell used to display the Box's title.

contentView

– **contentView**

Returns the Box's content view. The content view is created automatically when the Box is created, and resized as the Box is resized (you should never send frame-altering messages directly to a Box's content view). You can replace it with a View of your own through the **setContentView:** method.

See also: – **setContentView:**

drawSelf::

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

Fills the Box's background with opaque, light gray (NX_LTGRAY) paint, then draws the object's title, border, and its subviews (you can't change the background color short of creating your own Box subclass). You never invoke this method directly; it's invoked by the display methods inherited from the View class. Returns **self**.

font

– **font**

Returns the Font object used to draw the Box's title. By default, the Font is the 12.0 point system font (NXSystemFont).

See also: – **setFont:**

free

– **free**

Frees the Box and all its subviews.

getOffsets:

– **getOffsets:**(NXSize *)*theSize*

Returns, by reference in *theSize*, the horizontal and vertical distances between the Box's border and its content area, measured in the Box's coordinate system. The default is 5.0 in both dimensions. Returns **self**.

See also: – **setOffsets::**

initFrame:

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

The designated initializer for the Box class, this method initializes the Box with the following values:

Attribute	Value
frame	*frameRect*
title	"Title"
border type	NX_RIDGE
title position	NX_ATTOP
font	12.0 point NXSystemFont
offsets	5.0 in both dimensions

In addition, the Box's content view is automatically created and added as the Box's single subview, and the Box identifies itself as an opaque View. Returns **self**.

read:

– **read:**(NXTypedStream *)*stream*

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

See also: – **write:**

replaceSubview:with:

– **replaceSubview:***oldView* **with:***newView*

Replace *oldView* with *newView* in the subview list of the Box's content view. This method does nothing and returns **nil** if *oldView* isn't a subview of the content view or if *newView* isn't a View. Otherwise, this method returns *oldView*.

See also: – **addSubview:**

setBorderType:

– **setBorderType:**(int)*aType*

Sets the border type to *aType*, which must be NX_LINE, NX_GROOVE, NX_BEZEL, or NX_NOBORDER (a Box's default border type is NX_GROOVE). If the size of the new border is different from that of the old border, the content view is resized to absorb the difference. The Box isn't redisplayed. Returns **self**.

See also: – **borderType**

setContentView:

– **setContentView:***aView*

Sets the Box's content view to *aView*, resizing the View to fit within the Box's current content area. The old content view is returned.

See also: – **contentView**

setFont:

– **setFont:***fontObj*

Sets *fontObj* as the Font object used to draw the Box's title. By default, the title is drawn using the 12.0 point system font (NXSystemFont). If the size of the new Font is different from that of the old Font, the content view is resized to absorb the difference. The Box isn't redisplayed. Returns **self**.

See also: + **newFont:size:** (Font)

setFrameFromContentFrame:

– **setFrameFromContentFrame:**(const NXRect *)*contentFrame*

Places the Box so its content view lies on *contentFrame*, reckoned in the coordinate system of the Box's superview. Returns **self**.

See also: – **setOffsets::**, – **setFrame:** (View)

setOffsets::

– **setOffsets:**(NXCoord)*horizontal* **:**(NXCoord)*vertical*

Sets the horizontal and vertical distance between the border of the Box and its content view. The *horizontal* value is applied (reckoned in the Box's coordinate system) fully and equally to the left and right sides of the Box. The *vertical* value is similarly applied to the top and bottom. Returns **self**

Unlike changing a Box's other attributes, such as its title position or border type, changing the offsets *doesn't* automatically resize the content view. In general, you should send a **sizeToFit** message to the Box after changing the size of its offsets. This causes the content view to remain unchanged while the Box is wrapped around it.

setTitle:

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

Sets the title to *aString*. By default, a Box's title is "Title". After invoking this method you should send a **sizeToFit** message to the Box to ensure that it's wide enough to accommodate the length of the title. Returns **self**.

See also: – **title**, – **setFont:**

setTitlePosition:

– **setTitlePosition:**(int)*aPosition*

Sets the title position to *aPosition*, which can be one of the values listed in the following table. The default position is NX_ATTOP.

Value	Meaning
NX_NOTITLE	The Box has no title
NX_ABOVETOP	Title positioned above the Box's top border
NX_ATTOP	Title positioned within the Box's top border
NX_BELOWTOP	Title positioned below the Box's top border
NX_ABOVEBOTTOM	Title positioned above the Box's bottom border
NX_ATBOTTOM	Title positioned within the Box's bottom border
NX_BELOWBOTTOM	Title positioned below the Box's bottom border

If the new title position changes the size of the Box's border area, the content view is resized to absorb the difference. The Box isn't redisplayed. Returns **self**.

See also: – **getTitlePosition:**

sizeTo::

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

Resizes the Box to *width* and *height*. The Box's content view is resized accordingly; if the new width or height of the Box leaves no room for the content view (after subtracting the room needed to accommodate the border, title, and offsets), the respective dimension of the content view will be zero. Returns **self**.

sizeToFit

– **sizeToFit**

Resizes and moves the Box's content view so that it just encloses its subviews. The Box itself is then moved and resized to wrap around the content view. The Box's width is constrained so its title will be fully displayed.

You should invoke this method after:

- Adding a subview (to the content view).
- Altering the size or location of such a subview.
- Setting the Box's offsets.
- Setting the Box's title.

The mechanism by which the content view is moved and resized depends on whether the object responds to its own **sizeToFit** message: If it does respond, then that message is sent and the content view is expected to be so modified. If the content view doesn't respond, the Box moves and resizes the content view itself.

Returns **self**.

title

– (const char *)**title**

Returns the Box's title. By default, a Box's title is "Title".

See also: – **setTitle:**

titlePosition

– (int)**titlePosition**

Returns a constant representing the title position. See the description of **setTitlePosition:** for a list of the title position constants.

See also: – **setTitlePosition:**

write:

– **write:**(NXTypedStream *)*stream*

Writes the Box to the typed stream *stream*. Returns **self**.

See also: – **read:**

Button

Inherits From: Control : View : Responder : Object

Declared In: appkit/Button.h

Class Description

Button is a subclass of Control that intercepts mouse-down events and sends an action message to a target object when it's clicked or pressed. By virtue of its ButtonCell, Button is a two-state Control—it's either "off" or "on"—and it displays its state depending on the configuration of the ButtonCell. Button acquires other attributes of ButtonCell. The state is used as the value, so Control methods like **setIntValue:** actually set the state (the methods **setState:** and **state** are provided for more conceptually accurate setting of the state). The Button can send its action continuously and display highlighting in several different ways. What's more, a Button can have a key equivalent that's eligible for triggering whenever the Button's Panel or Window is key.

Button and Matrix both provide a Control View needed to display a ButtonCell object. However, while Matrix requires you to access the ButtonCells directly, most of Button's methods are covers for identically declared methods in ButtonCell. The only ButtonCell methods that don't have covers relate to the font used to display the key equivalent, and to specific methods for highlighting or showing the Button's state (these last are usually set together with Button's **setType:** method).

Creating a Subclass of Button

The **initFrame:icon:tag:target:action:key:enabled:** method is the designated initializer for Buttons that initially display only icons. Buttons that initially display only text have the designated initializer **initFrame:text:tag:target:action:key:enabled:**. Override one or both of these methods if you create a subclass of Button that performs its own initialization.

In particular, if you want to use a custom ButtonCell subclass with your subclass of Button, you have to override the **setCellClass:** method as well as the designated initializers, as described in "Creating New Controls" in the Control class specification.

See the ButtonCell class specification for more on Button's behavior.

Instance Variables

None declared in this class.

Method Types

Setting Button's Cell class	+ setCellClass:
Initializing a Button	– init – initFrame: – initFrame:icon:tag:target:action:key:enabled: – initFrame:title:tag:target:action:key:enabled:
Setting the Button type	– setType:
Setting the state	– setState: – state
Setting the repeat interval	– setPeriodicDelay:andInterval: – getPeriodicDelay:andInterval:
Setting the titles	– setTitle: – setTitleNoCopy: – title – setAltTitle: – altTitle
Setting the icons	– setIcon: – setIcon:position: – icon – setAltIcon: – altIcon – setImage: – image – setAltImage: – altImage – setIconPosition: – iconPosition
Modifying graphic attributes	– setTransparent: – isTransparent – setBordered: – isBordered
Displaying the Button	– display – highlight:

Setting the key equivalent	– setKeyEquivalent: – keyEquivalent
Handling events and action messages	– acceptsFirstMouse – performClick: – performKeyEquivalent:
Setting the Sound	– setSound: – sound

Class Methods

setCellClass:

+ **setCellClass:***classId*

Configures the Button class to use instances of *classId* for its Cells. *classId* should be the **id** of a subclass of ButtonCell, obtained by sending the **class** message to either the Cell subclass object or to an instance of that subclass. The default Cell class is ButtonCell. Returns **self**.

If this method isn't overridden by a subclass of Button, then when it's sent to that subclass, Button and any other subclasses of Button that don't override the methods mentioned below will use the new Cell subclass as well. To safely set a Cell class for your subclass of Button, override this method to store the Cell class in a static **id**. Also, override the designated initializer to replace the Button subclass instance's Cell with an instance of the Cell subclass stored in that static **id**. See "Creating New Controls" in the Control class specification's class description for more information.

Instance Methods

acceptsFirstMouse

– (BOOL)**acceptsFirstMouse**

Returns YES. Buttons always accept the mouse-down event that activates a Window, regardless of whether the Button is enabled.

altIcon

– (const char *)**altIcon**

Returns the name of the NXImage that appears on the Button when it's in its alternate state, or NULL if there is no alternate icon or the NXImage has no name. This NXImage is displayed only for Buttons that highlight or show their alternate state by displaying their alternate contents (as opposed to simply lighting or pushing in).

See also: – **setAltIcon:**, – **setIconPosition:**, – **altImage**, – **icon**, – **image**, – **setType:**

altImage

– **altImage**

Returns the NXImage that appears on the Button when it's in its alternate state, or **nil** if there is no alternate NXImage. This Button only displays its alternate NXImage if it highlights or shows its alternate state by displaying its alternate contents.

See also: – **setAltImage:**, – **setIconPosition:**, – **altIcon**, – **image**, – **icon**, – **setType:**

altTitle

– (const char *)**altTitle**

Returns the string that appears on the Button when it's in its alternate state, or NULL if there isn't one. The alternate title is only displayed if the Button highlights or shows its alternate state by displaying its alternate contents.

See also: – **setAltTitle:**, – **title**, – **setType:**

display

– **display**

Displays the Button. This method is overridden from View so that **displayFromOpaqueAncestor:::** is invoked if the Button is not opaque. Returns **self**.

See also: – **isOpaque** (Cell), – **isTransparent**, – **setTransparent:**

getPeriodicDelay:andInterval:

– **getPeriodicDelay:**(float *)*delay* **andInterval:**(float *)*interval*

Returns **self**, and by reference the delay and interval periods for a continuous Button. *delay* is the amount of time (in seconds) that a continuous Button will pause before starting to periodically send action messages to the target object. *interval* is the amount of time (also in seconds) between those messages.

See also: – **setContinuous:** (Control), – **setPeriodicDelay:andInterval:**

highlight:

– **highlight:**(BOOL)*flag*

If the highlight state of the cell is not equal to *flag*, the Button is highlighted and the highlight state of the cell is set to *flag*. Highlighting may involve the Button appearing "pushed in" to the screen, displaying its alternate title or icon, or lighting. This method issues a **flushWindow** message after highlighting the Button. Returns **self**.

See also: – **setType:**

icon

– (const char *)**icon**

Returns the name of the NXImage that appears on the Button when it's in its normal state, or NULL if there is no such NXImage or the NXImage doesn't have a name. A Button that doesn't display its alternate contents to highlight or show its alternate state will always display its normal icon.

See also: – **setIcon:**, – **setIcon:position:**, – **setIconPosition:**, – **image**, – **altIcon**, – **altImage**, – **setType:**

iconPosition

– (int)**iconPosition**

Returns the position of the icon (if any) on the Button. See **setIconPosition:** for the list of positions.

See also: – **setIconPosition:**, – **setIcon:position:**

image

– **image**

Returns the NXImage that appears on the Button when it's in its normal state, or **nil** if there is no such NXImage. This NXImage is always displayed on a Button that doesn't change its contents when highlighting or showing its alternate state.

See also: – **setImage:**, – **setIconPosition:**, – **icon**, – **altImage**, – **altIcon**, – **setType:**

init

– **init**

Initializes and returns the receiver, a new Button instance, with a frame origin of (0, 0) and width and height of 50 units each. The new instance is enabled and displays the default title "Button" centered in its frame, but has no icon, tag, target, action, or key equivalent associated with it. The new Button is bordered, and is of type NX_MOMENTARYPUSH. One of the more specific initializers is usually used to initialize a Button.

See also: – **initFrame:title:tag:target:action:key:enabled:**, – **initFrame:icon:tag:target:action:key:enabled:**, – **initFrame:**, – **setType:**

initFrame:

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

Initializes and returns the receiver, a new Button instance, with default parameters in the given frame. The new instance is enabled and displays the default title "Button" centered in its frame, but has no icon, tag, target, action, or key equivalent. The new Button is bordered, and is of type NX_MOMENTARYPUSH. One of the more specific initializers is usually used to initialize a Button.

See also: – **initFrame:title:tag:target:action:key:enabled:**, – **initFrame:icon:tag:target:action:key:enabled:**, – **initFrame:**, – **setType:**

initFrame:icon:tag:target:action:key:enabled:

– **initFrame:**(const NXRect *)*frameRect*
icon:(const char *)*iconName*
tag:(int)*anInt*
target:*anObject*
action:(SEL)*aSelector*
key:(unsigned short)*charCode*
enabled:(BOOL)*flag*

Initializes and returns the receiver, a new Button instance that displays an icon. *frameRect* is the rectangle defining the Button's position and size in its superview. *iconName* is the name of an NXImage that will be used for the Button's icon. *anInt* is set as the Button's tag. *anObject* is set as the target, which will be sent *aSelector* when the Button is clicked or pressed. *charCode* is the new Button's key equivalent. *flag* determines whether the Button is enabled or not. The new Button is bordered, and is of type NX_MOMENTARYPUSH.

This method is the designated initializer for Buttons that display icons. A Button that displays an icon can be configured to also display a title with the **setTitle:** and **setIconPosition:** methods.

See also: – **setTitle:**, – **setIconPosition:**, – **setType:**

initFrame:title:tag:target:action:key:enabled:

– **initFrame:**(const NXRect *)*frameRect*
title:(const char *)*aString*
tag:(int)*anInt*
target:*anObject*
action:(SEL)*aSelector*
key:(unsigned short)*charCode*
enabled:(BOOL)*flag*

Initializes and returns the receiver, a new Button instance that displays a text string. The arguments and operation of this method are identical to those of **initFrame:icon:tag:target:action:key:enabled:**, except that *aString* is the title that the Button will display instead of the name of an icon. The new Button is bordered, and is of type NX_MOMENTARYPUSH.

This method is the designated initializer for Buttons that display text. A Button that displays an icon can be configured to also display an icon with the **setIcon:position:** method, or a combination of **setIcon:** or **setImage:** and **setIconPosition:**.

See also: – **setIcon:**, – **setImage:**, – **setIconPosition:**, – **setType:**

isBordered

– (BOOL)**isBordered**

Returns YES if the Button has a border, NO otherwise. A Button's border isn't the single line of most other Controls' borders; instead, it's a raised bezel ("bezel" usually refers to a depressed bezel, as seen on FormCells, for example). You shouldn't use the **setBezeled:** method with a Button.

See also: – **setBordered:**

isTransparent

– (BOOL)**isTransparent**

Returns YES if the Button is transparent, NO otherwise. A transparent Button never draws itself, but it receives mouse-down events and tracks the mouse properly.

See also: – **setTransparent:**

keyEquivalent

– (unsigned short)**keyEquivalent**

Returns the key equivalent character of the Button, or 0 if one hasn't been defined.

See also: – **setKeyEquivalent:**, – **performKeyEquivalent:**

performClick:

– **performClick:***sender*

Highlights the Button, sends its action message to the target object, then unhighlights the Button. Invoke this method when you want the Button to behave exactly as if the user had clicked it with the mouse.

See also: – **performKeyEquivalent:**

performKeyEquivalent:

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

If the character in *theEvent* matches the Button's key equivalent, simulates the user clicking the Button by sending **performClick:** to **self**, and returns YES. Otherwise, does nothing and returns NO.

The Button won't perform the key equivalent if there's a modal panel present that the Button isn't on.

See also: – **keyEquivalent**, – **performClick:**

setAltIcon:

– **setAltIcon:**(const char *)*iconName*

Sets the Button's alternate icon by name; *iconName* is the name of the NXImage to be displayed. Doesn't display the Button even if autodisplay is on. Returns **self**.

A Button's alternate icon is only displayed if the Button highlights or shows its alternate state by changing its contents.

See also: – **altIcon**, – **setIconPosition:**, – **setAltImage:**, – **setIcon:**, – **setImage:**, + **findImageNamed**: (NXImage), – **setType:**, – **setAutodisplay:** (View)

setAltImage:

– **setAltImage:***altImage*

Sets the Button's alternate icon by **id**; *altImage* is the NXImage to be displayed. Returns **self**.

A Button displays its alternate NXImage only if it highlights or displays its alternate state by using its alternate contents.

See also: – **altImage**, – **setIconPosition:**, – **setAltIcon:**, – **setImage:**, – **setIcon:**, – **setType:**

setAltTitle:

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

Sets the title that the Button displays in its alternate state to *aString*. Returns **self**.

The alternate title is shown only if the Button changes its contents when highlighting or displaying its alternate state.

See also: – **altTitle:**, – **setTitle:**, – **setType:**

setBordered:

– **setBordered:**(BOOL)*flag*

If *flag* is YES, the Button displays a border; if NO, the Button doesn't display a border. A Button's border is not the single line or most other Controls' borders; instead, it's a raised bezel ("bezel" usually refers to a depressed bezel, as seen on FormCells, for example). This method redraws the Button if the bordered state changes. Returns **self**.

See also: – **isBordered**

setIcon:

– **setIcon:**(const char *)*iconName*

Sets the Button's icon by name; *iconName* is the name of the NXImage to be displayed. Redraws the Button's inside and returns **self**.

A Button's icon is displayed when the Button is in its normal state, or always if the Button doesn't highlight or show state by changing its contents.

See also: – **setIcon:position:**, – **icon**, – **setIconPosition:**, – **setImage:**, – **setAltIcon:**, – **setAltImage:**, + **findImageNamed:** (NXImage), – **setType:**

setIcon:position:

– **setIcon:**(const char *)*iconName* **position:**(int)*aPosition*

Combines **setIcon:** and **setIconPosition:** into one message. Returns **self**.

See also: – **setIcon:**, – **setIconPosition:**

setIconPosition:

– **setIconPosition:**(int)*aPosition*

Sets the position of the icon when a Button simultaneously displays both text and an icon. *aPosition* can be one of the following constants:

NX_TITLEONLY	title only (no icon on the Button)
NX_ICONONLY	icon only (no text on the Button)
NX_ICONLEFT	icon is to the left of the text
NX_ICONRIGHT	icon is to the right of the text
NX_ICONBELOW	icon is below the text
NX_ICONABOVE	icon is above the text
NX_ICONOVERLAPS	icon and text overlap (text drawn over icon)

If the position is top or bottom, the alignment of the text will be changed to NX_CENTERED. This behavior can be overridden with a subsequent **setAlignment:** method. Redraws the Button's inside and returns **self**.

See also: – **iconPosition**, – **setIcon:position:**, – **setAlignment:** (Control)

setImage:

– **setImage:***image*

Sets the Button's icon by **id**; *image* is the NXImage to be displayed. Redraws the Button's inside and returns **self**.

A Button's icon is displayed when the Button is in its normal state, or all the time for a Button that doesn't change its contents when highlighting or displaying its alternate state.

See also: – **image**, – **setIconPosition:**, – **setIcon:**, – **setAltImage:**, – **setAltIcon:**, – **setType:**

setKeyEquivalent:

– **setKeyEquivalent:**(unsigned short)*charCode*

Sets the key equivalent character of the Button, and redraws the Button's inside if there is no icon or alternate icon set for the Button. The key equivalent isn't displayed if the icon position is set to NX_TITLEONLY, NX_ICONONLY or NX_ICONOVERLAPS; that is, the Button must display both its title and its "icon" (the key equivalent in this case), and they must not overlap. Returns **self**.

To display a key equivalent on a Button, set the image and alternate image to **nil**, then set the key equivalent, and then set the icon position.

See also: – **keyEquivalent**, – **setIconPosition:**, – **performKeyEquivalent:**, – **setImage:**, – **setAltImage:**

setPeriodicDelay:andInterval:

– **setPeriodicDelay:**(float)*delay* **andInterval:**(float)*interval*

Sets the message delay and interval for the Button. These two values are used if the Button is configured (by a **setContinuous:** message) to continuously send the action message to the target object while tracking the mouse. *delay* is the amount of time (in seconds) that a continuous Button will pause before starting to periodically send action messages to the target object. *interval* is the amount of time (also in seconds) between those messages. Returns **self**.

The maximum value allowed for both the delay and the interval is 60.0 seconds.

See also: – **getPeriodicDelay:andInterval:**, – **setContinuous** (Control)

setSound:

– **setSound:***soundObject*

Sets the Sound played when the Button is pressed, and whenever the cursor re-enters the Button while tracking. Returns **self**.

See also: – **sound**

setState:

– **setState:**(int)*anInt*

Sets the Button's state to *anInt* and redraws the Button. 0 is the normal or "off" state, and any nonzero number is the alternate or "on" state. Returns **self**.

See also: – **state**

setTitle:

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

Sets the title displayed by the Button when in its normal state to *aString*. This title is always shown on Buttons that don't use their alternate contents when highighting or displaying their alternate state. Redraws the Button's inside and returns **self**.

See also: – **setTitleNoCopy:**, – **title**, – **setAltTitle:**, – **setType:**

setTitleNoCopy:

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

Similar to **setTitle:** but doesn't make a copy of *aString*. Returns **self**.

See also: – **setTitle:**

setTransparent:

– **setTransparent:**(BOOL)*flag*

Sets whether the Button is transparent, and redraws the Button if *flag* is NO. Returns **self**.

A transparent Button tracks the mouse and sends its action, but doesn't draw. A transparent Button is useful for sensitizing an area on the screen so that an action gets sent to a target when the area receives a mouse click.

See also: – **isTransparent**

setType:

– **setType:**(int)*aType*

Sets the way the Button highlights while pressed, and how it shows its state. Redraws the Button and returns **self**. The types available are for the most common Button types, which are also accessible in Interface Builder; you can configure different behavior with ButtonCell's **setHighlightsBy:** and **setShowsStateBy:** methods. *aType* can be one of seven constants:

NX_MOMENTARYPUSH (the default): While the Button is held down it's shown as lit, and also "pushed in" to the screen if the Button is bordered. This type of Button is best for simply triggering actions, as it doesn't show its state; it always displays its normal icon or title. This option is called "Momentary Push" in Interface Builder's Button Inspector.

NX_MOMENTARYCHANGE: While the Button is pressed, the alternate icon or alternate title is displayed. This type always displays its normal title or icon (that is, it doesn't display its state). The miniaturize button in a window's title bar is a good example of this type of Button. This option is called "Momentary Change" in Interface Builder's Button Inspector.

NX_PUSHONPUSHOFF: Holding the Button down causes it to be shown as lit, and also "pushed in" to the screen if the Button is bordered. The Button displays itself as lit while in its alternate state. This option is called "Push On/Push Off" in Interface Builder's Button Inspector.

NX_ONOFF: Highlights while pressed by lighting, and stays lit in its alternate state. This option is called "On/Off" in Interface Builder's Button Inspector.

NX_TOGGLE: Highlighting is performed by changing to the alternate title or icon "pushing in." The alternate state is shown by displaying the alternate title or icon. This option is called "Toggle" in Interface Builder's Button Inspector.

NX_SWITCH: A variant of NX_TOGGLE that has no border, and that has a default icon called "switch" and an alternate icon called "switchH" (these are identical to the "NXswitch" and "NXswitchH" system bitmaps). This type of Button is available as a separate palette item in Interface Builder.

NX_RADIOBUTTON: Like NX_SWITCH, but the default icon is "radio" and the alternate icon is "radioH" (identical to the "NXradio" and "NXradioH" system bitmaps). This type of Button is available as a separate palette item in Interface Builder.

There is no constant for Interface Builder's "Momentary Light" type; you can set this programmatically as follows:

```
[[myButton cell] setHighlightsBy:NX_CHANGEGRAY | NX_CHANGEBACKGROUND];
[[myButton cell] setShowsStateBy:NX_NONE];
```

See also: – **setType:** (ButtonCell), – **setHighlightsBy:** (ButtonCell), – **setShowsStateBy:** (ButtonCell)

sound

– sound

Returns the Sound played when the Button is pressed, and whenever the cursor re-enters the Button while tracking.

See also: – **setSound:**

state

– (int)**state**

Returns the Button's state, either 0 for normal or "off," or 1 for alternate or "on."

See also: – **setState:**

title

– (const char *)**title**

Returns the title displayed on the Button when it's in its normal state, or always if the Button doesn't use its alternate contents for highlighting or displaying the alternate state. Returns NULL if there is no title.

See also: – **setTitle:**, – **altTitle**, – **setType:**

ButtonCell

Inherits From: ActionCell : Cell : Object

Declared In: appkit/ButtonCell.h

Class Description

ButtonCell is a subclass of ActionCell used to implement the user interface devices of push buttons, switches, and radio buttons, as well as any area of the screen that should send a message to a target when clicked. ButtonCells are used by the Button and Matrix subclasses of Control. Matrix is specifically used to hold sets of ButtonCells to create groups of switches or radio buttons.

A ButtonCell is a two-state Cell; it's either "off" or "on," and can be configured to display the two states differently, with a separate title and/or icon (named NXImage) for either state. The two states are more often referred to as "normal" and "alternate." A ButtonCell's state is also used as its value, so Cell methods that set the value (**setIntValue:** and so on) actually set the ButtonCell's state to "on" if the value provided is non-zero (or non-null for strings), and to "off" if the value is zero or null. Similarly, methods that retrieve the value return 1 for the "on" or alternate state (an empty string in the case of **stringValue**), or 0 or NULL for the "off" or normal state. Unlike Button, ButtonCell doesn't have **setState:** or **state** methods; you have to use **setIntValue:** or a related method.

A ButtonCell sends its action message to its target once if it's View is clicked and it gets the mouse-down event, but can also send the action message continuously as long as the mouse is held down with the cursor inside the ButtonCell. The ButtonCell can show that it's being pressed by highlighting in several ways, for example, a bordered ButtonCell can appear pushed into the screen, or the icon or title can change to an alternate form while the ButtonCell is pressed.

A ButtonCell can also have a key equivalent (like a Menu item). If the Window or Panel that the ButtonCell's View is on is the key window, then it gets the first chance to receive events related to key equivalents. This is used quite often in modal panels that have an "OK" Button with a Return sign on them. Usually a ButtonCell displays a key equivalent as its icon; if you ever set an icon for the ButtonCell, the key equivalent remains, but doesn't get displayed.

For more information on ButtonCell's behavior, see the Button and Matrix class specifications.

Instance Variables

```
char *altContents;
union _icon {
        struct _bmap {
                id normal;
                id alternate;
        } bmap;
        struct _ke {
                id font;
                float descent;
        } ke;
} icon;
id sound;
struct _bcFlags1 {
        unsigned int pushIn:1;
        unsigned int changeContents:1;
        unsigned int changeBackground:1;
        unsigned int  changeGray:1;
        unsigned int lightByContents:1;
        unsigned int lightByBackground:1;
        unsigned int lightByGray:1;
        unsigned int hasAlpha:1;
        unsigned int bordered:1;
        unsigned int iconOverlaps:1;
        unsigned int horizontal:1;
        unsigned int bottomOrLeft:1;
        unsigned int iconAndText:1;
        unsigned int lastState:1;
        unsigned int iconSizeDiff:1;
        unsigned int iconIsKeyEquivalent:1;
} bcFlags1;
struct _bcFlags2 {
        unsigned int keyEquivalent:8;
        unsigned int transparent:1;
} bcFlags2;
unsigned short periodicDelay;
unsigned short periodicInterval;
```

altContents	The contents shown when the ButtonCell is in its alternate state: a string for a text ButtonCell, an NXImage for an icon-only ButtonCell.
icon.bmap.normal	The icon for a ButtonCell that displays both a title and an icon.
icon.bmap.alternate	The alternate icon for a ButtonCell that displays both a title and an icon.
icon.ke.font	Font used to draw the key equivalent.
icon.ke.descent	The descent of descenders in the key equivalent font.
sound	The Sound played when the ButtonCell gets a mouse-down event.
bcFlags1.pushIn	If 1, a bordered ButtonCell appears to push into the screen when pressed.
bcFlags1.changeContents	If 1, the ButtonCell shows its alternate state by displaying its alternate icon and title.
bcFlags1.changeBackground	If 1, the ButtonCell shows its alternate state by swapping the light gray and white pixels in its background.
bcFlags1.changeGray	If 1, the ButtonCell shows its alternate state by swapping its light gray and white pixels.
bcFlags1.lightByContents	If 1, the ButtonCell highlights while pressed by displaying its alternate icon and title.
bcFlags1.lightByBackground	If 1, the ButtonCell highlights by swapping the light gray and white pixels in its background.
bcFlags1.lightByGray	If 1, the ButtonCell shows its highlighting by swapping its light gray and white pixels.
bcFlags1.hasAlpha	1 if the ButtonCell's icon has alpha values (transparent pixels).
bcFlags1.bordered	1 if the ButtonCell has a raised bezel border.
bcFlags1.iconOverlaps	1 if the icon overlaps the title.
bcFlags1.horizontal	1 if the icon is to one side of title.
bcFlags1.bottomOrLeft	1 if the icon is on the left or bottom.
bcFlags1.iconAndText	1 if the ButtonCell has both an icon and a title.
bcFlags1.lastState	The state of the ButtonCell when last drawn.
bcFlags1.iconSizeDiff	1 if the alternate icon is a different size than the normal icon.

bcFlags1.iconIsKeyEquivalent	1 if the key equivalent is drawn as the icon.
bcFlags2.keyEquivalent	The key equivalent character.
bcFlags2.transparent	1 if the ButtonCell doesn't draw itself at all.
periodicDelay	The delay before sending the first action message by a continuous ButtonCell.
periodicInterval	The interval at which a continuous ButtonCell sends its action.

Method Types

Initializing, copying, and freeing a ButtonCell
- init
- initTextCell:
- initIconCell:
- copyFromZone:
- free

Determining component sizes
- calcCellSize:inRect:
- getDrawRect:
- getTitleRect:
- getIconRect:

Setting the titles
- setTitle:
- setTitleNoCopy:
- title
- setAltTitle:
- altTitle
- setFont:

Setting the icons
- setIcon:
- icon
- setAltIcon:
- altIcon
- setImage:
- image
- setAltImage:
- altImage
- setIconPosition:
- iconPosition

Setting the Sound
- setSound:
- sound

Setting the state	– setDoubleValue: – doubleValue – setFloatValue: – floatValue – setIntValue: – intValue – setStringValue: – setStringValueNoCopy: – stringValue
Setting the repeat interval	– setPeriodicDelay:andInterval: – getPeriodicDelay:andInterval:
Tracking the mouse	– trackMouse:inRect:ofView:
Setting the key equivalent	– setKeyEquivalent: – setKeyEquivalentFont: – setKeyEquivalentFont:size: – keyEquivalent
Setting parameters	– setParameter:to: – getParameter:
Modifying graphic attributes	– setBordered: – isBordered – setTransparent: – isTransparent – isOpaque
Modifying display behavior	– setType: – setHighlightsBy: – highlightsBy – setShowsStateBy: – showsStateBy
Simulating a click	– performClick:
Displaying the ButtonCell	– drawInside:inView: – drawSelf:inView: – highlight:inView:lit:
Archiving	– read: – write:

Instance Methods

altIcon

– (const char *)**altIcon**

Returns the name of the NXImage that appears on the ButtonCell when it's in its alternate state, or NULL if there is no alternate icon or the NXImage has no name. This NXImage is displayed only for ButtonCells that highlight or show their alternate state by displaying their alternate contents (as opposed to simply lighting or pushing in).

See also: – **setAltIcon:**, – **setIconPosition:**, – **altImage**, – **icon**, – **image**, – **setType:**

altImage

– **altImage**

Returns the NXImage that appears on the ButtonCell when it's in its alternate state, or **nil** if there is no alternate NXImage. This ButtonCell only displays its alternate NXImage if it highlights or shows its alternate state by displaying its alternate contents.

See also: – **setAltImage:**, – **setIconPosition:**, – **altIcon**, – **image**, – **icon**, – **setType:**

altTitle

– (const char *)**altTitle**

Returns the string that appears on the ButtonCell when it's in its alternate state, or NULL if there isn't one. The alternate title is only displayed if the ButtonCell highlights or shows its alternate state by displaying its alternate contents.

See also: – **setAltTitle:**, – **title**, – **setType:**

calcCellSize:inRect:

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

Returns **self**, and by reference in *theSize* the minimum width and height required for displaying the ButtonCell in *aRect*. This minimum size is the larger of the sizes required for displaying the normal contents or the alternate contents, plus any space needed to display a border.

See also: – **getDrawRect:**, – **getIconRect:**, – **getTitleRect:**

copyFromZone:

– **copyFromZone:**(NXZone *)*zone*

Allocates, initializes, and returns a copy of the receiving ButtonCell. The copy is allocated from *zone* and is given the same data as the receiver.

doubleValue

– (double)**doubleValue**

Returns 0.0 if the ButtonCell is in its normal state, 1.0 if the it's in its alternate state.

See also: – **setDoubleValue:**, – **floatValue**, – **intValue**, – **stringValue**

drawInside:inView:

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

Draws the inside of the ButtonCell (the title, icon, and their background, but not the border) in *aRect* within *controlView*. *aRect* should be the same rectangle passed to **drawSelf:inView:**. The PostScript focus must be locked on *controlView* when this message is sent. This method is invoked by **drawSelf:inView:** and by the Control classes' **drawCellInside:** method. It's provided so that when a ButtonCell's state is set (with **setIntValue:**, for example), a minimal update of the ButtonCell's visual appearance can occur. Returns **self**.

If you subclass ButtonCell and override **drawSelf:inView:**, you must also override this method. However, you are free to override only this method and not **drawSelf:inView:** if your subclass doesn't draw outside the area that ButtonCell draws in.

See also: – **drawInside:inView:** (Cell), – **drawSelf:inView:**, – **lockFocus** (View)

drawSelf:inView:

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

Displays the ButtonCell in *cellFrame* within *controlView*. The PostScript focus must be locked on *controlView* when this message is sent. Draws the border of the ButtonCell if necessary, then invokes **drawInside:inView:**. Returns **self**.

See also: – **drawInside:inView:**, – **lockFocus** (View)

floatValue

– (float)**floatValue**

Returns 0.0 if the ButtonCell is in its normal state, 1.0 if it's in its alternate state.

See also: – **setFloatValue:**, – **doubleValue**, – **intValue**, – **stringValue**

free

– **free**

Frees the memory used by the ButtonCell and returns **nil**.

getDrawRect:

– **getDrawRect:**(NXRect *)*theRect*

Returns **self** and, by reference in *theRect* the bounds of the area into which the title and icon (not including the border) are drawn. You must pass the bounds of the ButtonCell in *theRect* (the same bounds calculated by **calcCellSize:inRect:** and passed to **drawSelf:inView:**). This method assumes that the ButtonCell is being drawn in a flipped View.

See also: – **getIconRect:**, – **getTitleRect:**, – **calcCellSize:inRect:**

getIconRect:

– **getIconRect:**(NXRect *)*theRect*

Returns **self** and, by reference in *theRect*, the bounds of the area into which the icon of the ButtonCell will be drawn. This will be the larger of the bounds for the normal and the alternate icons. If the ButtonCell has no icon, then *theRect* will be completely zeroed. You must pass the bounds of the ButtonCell in *theRect* (the same bounds calculated by **calcCellSize:inRect:** and passed to **drawSelf:inView:**). This method assumes that the ButtonCell is being drawn in a flipped View. Returns **self**.

See also: – **getTitleRect:**, – **getDrawRect:**, – **calcCellSize:inRect:**

getParameter:

– (int)**getParameter:**(int)*aParameter*

Returns the value of one of the frequently accessed flags for a ButtonCell. See **setParameter:to:** for a list of the parameters and corresponding methods. Since the parameters are also accessible through normal querying methods, you shouldn't need to use this method often.

See also: – **setParameter:to:**

getPeriodicDelay:andInterval:

– **getPeriodicDelay:**(float *)*delay* **andInterval:**(float *)*interval*

Returns **self**, and by reference the delay and interval periods for a continuous ButtonCell. *delay* is the amount of time (in seconds) that a continuous ButtonCell will pause before starting to periodically send action messages to the target object. *interval* is the amount of time (also in seconds) between those messages.

See also: – **setContinuous:** (Cell), – **setPeriodicDelay:andInterval:**

getTitleRect:

– **getTitleRect:**(NXRect *)*theRect*

Returns **self** and, by reference in *theRect* a copy of the bounds of the area into which the ButtonCell's title will be drawn. This will be the larger of the bounds for the normal and the alternate titles. If the ButtonCell has no title, then *theRect* will be completely zeroed. You must pass the bounds of the ButtonCell in *theRect* (the same bounds calculated by **calcCellSize:inRect:** and passed to **drawSelf:inView:**). This method assumes that the ButtonCell is being drawn in a flipped View.

See also: – **getIconRect:**, – **getDrawRect:**, – **calcCellSize:inRect:**

highlight:inView:lit:

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

Displays the ButtonCell in *cellFrame* if its highlight state is not equal to *flag*. The PostScript focus must be locked on *controlView* when this method is invoked. If *flag* is YES, the ButtonCell is displayed as highlighted. How this is done depends on how the ButtonCell has been configured; see the description of **setHighlightsBy:** for the possible manners of highlighting. This method does nothing if the ButtonCell is disabled or transparent. Returns **self**.

See also: – **lockFocus** (View)

highlightsBy

– (int)**highlightsBy**

Returns the logical OR of flags that indicate the way the ButtonCell highlights when it gets a mouse-down event. See **setHighlightsBy:** for the list of flags.

See also: – **setHighlightsBy:**, – **showStateBy**, – **setShowsStateBy:**

icon

– (const char *)**icon**

Returns the name of the NXImage that appears on the ButtonCell when it's in its normal state, or NULL if there is no such NXImage or the NXImage doesn't have a name. A ButtonCell that doesn't display its alternate contents to highlight or show its alternate state will always display its normal icon.

See also: – **setIcon:**, – **setIcon:position:**, – **setIconPosition:**, – **image**, – **altIcon**, – **altImage**, – **setType:**

iconPosition

– (int)**iconPosition**

Returns the position of the ButtonCell's icon (if any). See **setIconPosition:** for a list of the valid positions.

See also: – **setIconPosition:**

image

– **image**

Returns the NXImage that appears on the ButtonCell when it's in its normal state, or **nil** if there is no such NXImage. This NXImage is always displayed on a ButtonCell that doesn't change its contents when highlighting or showing its alternate state.

See also: – **setImage:**, – **setIconPosition:**, – **icon**, – **altImage**, – **altIcon**, – **setType:**

init

– **init**

Initializes and returns the receiver, a new text ButtonCell, with the title "Button" aligned in the center. The new ButtonCell is enabled, but has no icon, tag, target, action, or key equivalent associated with it. The new ButtonCell is bordered, and is of type NX_MOMENTARYPUSH.

See also: – **initIconCell:**, – **initTextCell:**

initIconCell:

– **initIconCell:**(const char *)*iconName*

Initializes and returns the receiver, a new ButtonCell instance that displays an icon. *iconName* is the name of an NXImage that will be used for the Button's icon. The new ButtonCell is enabled, bordered, and is of type NX_MOMENTARYPUSH.

This is the designated initializer for ButtonCells that display icons.

See also: – **initTextCell:**, – **init**

initTextCell:

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

Initializes and returns the receiver, a new ButtonCell instance that displays a title. *aString* is the title that will be used; it will be displayed in the user's default system font (as set with the Preferences application), 12.0 point size, and aligned in the center. The new ButtonCell is enabled, is bordered, and is of type NX_MOMENTARYPUSH.

This is the designated initializer for ButtonCells that display titles.

See also: – **initIconCell:**, – **init**

intValue

– (int)**intValue**

Returns 0 if the ButtonCell is in its normal state, 1 if in its alternate state.

See also: – **setIntValue:**, – **doubleValue**, – **floatValue**, – **stringValue**

isBordered

– (BOOL)**isBordered**

Returns YES if the ButtonCell has a border, NO if not. A ButtonCell's border isn't the single line of most other Cells' borders; instead, it's a raised bezel ("bezel" usually refers to a depressed bezel, as seen on FormCells, for example).

See also: – **setBordered:**

isOpaque

– (BOOL)**isOpaque**

Returns YES if the ButtonCell draws over every pixel in its frame, NO if not. The ButtonCell is opaque only if it is not transparent and if it has a border.

See also: – **isBordered**, – **setBordered:**, – **isTransparent**, – **setTransparent:**

isTransparent

– (BOOL)**isTransparent**

Returns YES if the ButtonCell is transparent, NO if not. A transparent ButtonCell never draws anything, but it does receive mouse-down events and track the mouse properly.

See also: – **setTransparent:**, – **isOpaque**

keyEquivalent

– (unsigned short)**keyEquivalent**

Returns the key equivalent character of the ButtonCell, or 0 if one hasn't been set.

See also: – **setKeyEquivalent:**, – **setKeyEquivalentFont:**,
– **setKeyEquivalentFont:size:**

performClick:

– **performClick:***sender*

If this ButtonCell is contained in a Control, then invoking this method causes the ButtonCell to act as if the user had clicked it.

read:

– **read:**(NXTypedStream *)*stream*

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

See also: – **write:**

setAltIcon:

– **setAltIcon:**(const char *)*iconName*

Sets the ButtonCell's alternate icon by name; *iconName* is the name of the NXImage to be displayed. Has the ButtonCell redrawn if possible, and returns **self**.

A ButtonCell's alternate icon is only displayed if the ButtonCell highlights or shows its alternate state by changing its contents.

See also: – **altIcon**, – **setIconPosition:**, – **setAltImage:**, – **setIcon:**, – **setImage:**, + **findImageNamed**: (NXImage), – **setType:**

setAltImage:

– **setAltImage:***altImage*

Sets the Button's alternate icon by **id**; *altImage* is the NXImage to be displayed. Has the ButtonCell redrawn if possible, and returns **self**.

A ButtonCell displays its alternate NXImage only if it highlights or displays its alternate state by using its alternate contents.

See also: – **altImage**, – **setIconPosition:**, – **setAltIcon:**, – **setImage:**, – **setIcon:**, – **setType:**

setAltTitle:

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

Sets the title that the ButtonCell displays in its alternate state to *aString*. Doesn't display the ButtonCell even if autodisplay is on in the ButtonCell's View. Returns **self**.

The alternate title is shown only if the ButtonCell changes its contents when highlighting or displaying its alternate state.

See also: – **altTitle:**, – **setTitle:**, – **setType:**

setBordered:

– **setBordered:**(BOOL)*flag*

If *flag* is YES, the ButtonCell displays a border; if NO, the If *flag* is YES, the ButtonCell displays a border; if NO, the ButtonCell doesn't display a border. A ButtonCell's border is not the single line or most other Cells' borders; instead, it's a raised bezel ("bezel" usually refers to a depressed bezel, as seen on FormCells, for example). Your code shouldn't use **setBezeled:** with a ButtonCell. This method redraws the ButtonCell if the bordered state changes. Returns **self**.

See also: – **isBordered**

setDoubleValue:

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

If aDouble is 0.0, sets the ButtonCell's state to 0 (the normal state); if *aDouble* is nonzero, sets it to 1 (the alternate state). Returns **self**.

See also: – **doubleValue**, – **setFloatValue:**, – **setIntValue:**, – **setStringValue:**

setFloatValue:

– **setFloatValue:**(float)*aFloat*

If aDouble is 0.0, sets the ButtonCell's state to 0 (the normal state); if *aDouble* is nonzero, sets it to 1 (the alternate state). Returns **self**.

See also: – **floatValue**, – **setDoubleValue:**, – **setIntValue:**, – **setStringValue:**

setFont:

– **setFont:***fontObject*

Sets the Font used to displaying the title and alternate title. Does nothing if the cell has no title or alternate title. Returns **self**.

If the ButtonCell has a key equivalent, its Font is not changed, but the key equivalent's Font size is changed to match the new title Font.

See also: – **setKeyEquivalentFont:**, – **setKeyEquivalentFont:size:**

setHighlightsBy:

– **setHighlightsBy:**(int)*aType*

Sets the way the ButtonCell highlights itself while pressed, and returns **self**. *aType* can be the logical OR of one or more of the following constants:

NX_PUSHIN (the default): The ButtonCell "pushes in" when pressed if it has a border.

NX_NONE: The ButtonCell doesn't change. This flag is ignored if any others are set in *aType*.

NX_CONTENTS: The ButtonCell displays its alternate icon and/or title.

NX_CHANGEGRAY: The ButtonCell swaps the light gray and white pixels on the its background and icon.

NX_CHANGEBACKGROUND: Same as NX_CHANGEGRAY, but only background pixels are changed.

If both NX_CHANGEGRAY and NX_CHANGEBACKGROUND are specified, both are recorded, but which behavior is used depends on the ButtonCell's icon. If there is no icon, or if the icon has no alpha (tranparency) data, NX_CHANGEGRAY is used. If the icon does have alpha data, NX_CHANGEBACKGROUND is used; this allows the gray/white swap of the background to show through the icon's transparent pixels.

See also: – **highlightsBy**, – **setShowsStateBy:**, – **showsStateBy**

setIcon:

– **setIcon:**(const char *)*iconName*

Sets the Button's icon by name; *iconName* is the name of the NXImage to be displayed. Redraws the Button's inside and returns **self**.

A ButtonCell's icon is displayed when the ButtonCell is in its normal state, or always if the ButtonCell doesn't highlight or show state by changing its contents.

See also: – **setIcon:position:**, – **icon**, – **setIconPosition:**, – **setImage:**, – **setAltIcon:**, – **setAltImage:**, + **findImageNamed:** (NXImage), – **setType:**

setIconPosition:

– **setIconPosition:**(int)*aPosition*

Sets the position of the icon when a ButtonCell simultaneously displays both text and an icon. *aPosition* can be one of the following constants:

NX_TITLEONLY	title only (no icon on the Button)
NX_ICONONLY	icon only (no text on the Button)
NX_ICONLEFT	icon is to the left of the text
NX_ICONRIGHT	icon is to the right of the text
NX_ICONBELOW	icon is below the text
NX_ICONABOVE	icon is above the text
NX_ICONOVERLAPS	icon and text overlap (text drawn over icon)

If the position is top or bottom, the alignment of the text will be changed to NX_CENTERED. This behavior can be overridden with a subsequent **setAlignment:** method. Redraws the Button's inside and returns **self**.

See also: – **iconPosition**, – **setAlignment:** (ActionCell)

setImage:

– **setImage:***image*

Sets the Button's icon by **id**; *image* is the NXImage to be displayed. Redraws the Button's inside and returns **self**.

A ButtonCell's NXImage is displayed when the ButtonCell is in its normal state, or all the time for a ButtonCell that doesn't change its contents when highlighting or displaying its alternate state.

See also: – **image**, – **setIconPosition:**, – **setIcon:**, – **setAltImage:**, – **setAltIcon:**, – **setType:**

setIntValue:

– **setIntValue:**(int)*anInt*

Sets the ButtonCell's state to 1 if *anInt* is nonzero, 0 otherwise. Returns **self**.

See also: – **intValue**, – **setDoubleValue:**, – **setFloatValue:**, – **setStringValue:**

setKeyEquivalent:

– **setKeyEquivalent:**(unsigned short)*charCode*

Sets the key equivalent character of the ButtonCell. Has the ButtonCell redrawn if needed. The key equivalent isn't displayed if the icon position is set to NX_TITLEONLY, NX_ICONONLY or NX_ICONOVERLAPS. Returns **self**.

The key equivalent isn't displayed on a ButtonCell that has an icon. To make sure it gets displayed, set the image and alternate image to **nil** before using this method.

See also: – **keyEquivalent**, – **setKeyEquivalentFont:**, – **setKeyEquivalentFont:size:**, – **performKeyEquivalent:** (Button, Matrix classes)

setKeyEquivalentFont:

– **setKeyEquivalentFont:***fontObject*

Sets the Font used to draw the key equivalent, and has the ButtonCell redrawn if possible. Does nothing if there is already an icon associated with this ButtonCell. The default Font is the same as that used to draw the title. Returns **self**.

See also: – **setKeyEquivalentFont:size:**

setKeyEquivalentFont:size:

– **setKeyEquivalentFont:**(const char *)*fontName* **size:**(float)*fontSize*

Sets by name and size the font used to draw the key equivalent, and has the ButtonCell redrawn if possible. Does nothing if there is already an icon associated with this ButtonCell. The default Font is the same as that used to draw the title. Returns **self**.

See also: – **setKeyEquivalentFont:**

setParameter:to:

– **setParameter:**(int)*aParameter* **to:**(int)*value*

Sets the value of one of a number of frequently accessed flags for a ButtonCell to *value*, and returns **self**. You don't normally need to use this method since all of these flags can be set through specific methods (for example, **setEnabled:**, **setHighlightsBy:**, and so on). The following table lists each constant used to identify a parameter with the methods for setting and retrieving the value for that parameter:

Parameter Constant	Equivalent Methods
NX_CELLDISABLED	**setEnabled:**, **isEnabled**
NX_CELLSTATE	**setState:**, **state**
NX_CELLHIGHLIGHTED	**highlight:inView:lit:**, **isHighlighted**
NX_CELLEDITABLE	**setEditable:**, **isEditable**
NX_BUTTONINSET	(none—see below)
NX_CHANGECONTENTS	**setShowsStateBy:**, **showsStateBy**
NX_CHANGEBACKGROUND	**setShowsStateBy:**, **showsStateBy**
NX_CHANGEGRAY	**setShowsStateBy:**, **showsStateBy**
NX_LIGHTBYCONTENTS	**setHighlightsBy:**, **highlightsBy**
NX_LIGHTBYBACKGROUND	**setHighlightsBy:**, **highlightsBy**
NX_LIGHTBYGRAY	**setHighlightsBy:**, **highlightsBy**
NX_PUSHIN	**setHighlightsBy:**, **highlightsBy**
NX_OVERLAPPINGICON	**setIconPosition:**, **iconPosition**
NX_ICONHORIZONTAL	**setIconPosition:**, **iconPosition**
NX_ICONONLEFTORBOTTOM	**setIconPosition:**, **iconPosition**
NX_ICONISKEYEQUIVALENT	(see below)

NX_BUTTONINSET is the inset of the ButtonCell's icon from its frame. You can find out if a ButtonCell's icon is its key equivalent by checking that both the **image** and **altImage** methods return **nil**, and that the **keyEquivalent** method returns a nonzero value. Your code should *never* set the NX_ICONISKEYEQUIVALENT parameter; always use the **setKeyEquivalent:** method, removing the ButtonCell's icon if necessary.

See also: – **getParameter:**, – **setKeyEquivalent:**

setPeriodicDelay:andInterval:

– **setPeriodicDelay:**(float)*delay* **andInterval:**(float)*interval*

Sets the message delay and interval for the ButtonCell. These two values are used if the ButtonCell has been set—by a **setContinuous:** message—to continuously send its action message to its target object while tracking the mouse. *delay* is the amount of time (in seconds) that a continuous ButtonCell will pause before starting to periodically send action messages to the target object. *interval* is the amount of time (also in seconds) between those messages. Returns **self**.

The maximum value allowed for both *delay* and the *interval* is 60.0 seconds.

See also: – **getPeriodicDelay:andInterval:**, – **setContinuous:** (Cell)

setShowsStateBy:

– **setShowsStateBy:**(int)*aType*

Sets the way the ButtonCell indicates its alternate state. *aType* should be the logical OR of one or more of the following constants:

NX_NONE (the default): The ButtonCell doesn't change. This flag is ignored if any others are set in *aType*.

NX_CONTENTS: The ButtonCell displays its alternate icon and/or title.

NX_CHANGEGRAY: The ButtonCell swaps the light gray and white pixels on its background and icon.

NX_CHANGEBACKGROUND: Same as NX_CHANGEGRAY, but only the background pixels are changed.

If both NX_CHANGEGRAY and NX_CHANGEBACKGROUND are specified, both are recorded, but the actual behavior depends on the ButtonCell's icon. If there is no icon, or if the icon has no alpha (tranparency) data, NX_CHANGEGRAY is used. If the icon exists and has alpha data, NX_CHANGEBACKGROUND is used; this allows the gray/white swap of the background to show through the icon's transparent pixels.

See also: – **showsStateBy**, – **setHighlightsBy:**, – **highlightsBy**

setSound:

– **setSound:***aSound*

Sets the Sound that will be played when the mouse goes down in the ButtonCell, and whenever the cursor re-enters the ButtonCell while tracking. Be sure to link against the Sound Kit if you use a Sound object. Returns **self**.

See also: – **sound**

setStringValue:

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

Sets the ButtonCell's state to 1 if *aString* is non-null (even if the string is empty), 0 otherwise. Returns **self**.

See also: – **setStringValueNoCopy:**, – **stringValue**

setStringValueNoCopy:

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

Sets the ButtonCell's state to 1 if *aString* is non-null (even if the string is empty), 0 otherwise. Returns **self**.

See also: – **setStringValue:**, – **stringValue**, – **setDoubleValue:**, – **setFloatValue:**, – **setIntValue:**

setTitle:

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

Sets the title displayed by the ButtonCell when in its normal state to *aString*. This title is always shown on ButtonCells that don't use their alternate contents when highighting or displaying their alternate state. Redraws the Button's inside and returns **self**.

See also: – **setTitleNoCopy:**, – **title**, – **setAltTitle:**

setTitleNoCopy:

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

Similar to **setTitle:** but does not make a copy of *aString*. Returns **self**.

See also: – **setTitle:**

setTransparent:

– **setTransparent:**(BOOL)*flag*

Sets whether the ButtonCell is transparent. Returns **self**.

A transparent ButtonCell never draws, but does track the mouse and send its action normally. A transparent ButtonCell is useful for sensitizing an area on the screen so that an action gets sent to a target when the area receives a mouse click.

See also: – **isTransparent**, – **isOpaque**

setType:

– **setType:**(int)*aType*

Sets the way the ButtonCell highlights while pressed, and how it shows its state. Redraws the ButtonCell if possible and returns **self**. *aType* can be one of the following constants (as described in the Button class specification's **setType:** method description):

NX_MOMENTARYPUSH
NX_MOMENTARYCHANGE
NX_PUSHONPUSHOFF
NX_ONOFF
NX_TOGGLE
NX_SWITCH
NX_RADIOBUTTON

See also: – **setType:** (Button), – **setHighlightsBy:**, – **setShowsStateBy:**

showsStateBy

– (int)**showsStateBy**

Returns the logical OR of flags that indicate the way the ButtonCell shows its alternate state. See **setShowsStateBy:** for the list of flags.

See also: – **setShowsStateBy:**, – **highlightsBy**, – **setHighlightsBy:**

sound

– **sound**

Returns the Sound played when the ButtonCell gets a mouse-down event, and whenever the cursor re-enters the ButtonCell while tracking.

See also: – **setSound:**

stringValue

– (const char *)**stringValue**

Returns "" (an empty string) if the ButtonCell's state is 1 (the alternate state), or NULL if the state is 0 (the normal state).

See also: – **setStringValue:**, – **setStringValueNoCopy:**, – **doubleValue**, – **floatValue**, – **intValue**

title

– (const char *)**title**

Returns the title displayed on the Button when it's in its normal state, or always if the Button doesn't use its alternate contents for highlighting or displaying the alternate state. Returns NULL if there is no title.

See also: – **setTitle:**, – **setTitleNoCopy:**

trackMouse:inRect:ofView:

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

Tracks the mouse by starting the Sound (if any) and sending **trackMouse:inRect:ofView** to **super** with the same arguments. When **super**'s method returns, stops the Sound if needed and returns YES if the mouse Button went up with the cursor in the cell, NO otherwise. This method returns if the cursor leaves the bounds of the ButtonCell.

See also: – **trackMouse:inRect:ofView:** (Cell)

write:

– **write:**(NXTypedStream *)*stream*

Writes the receiving ButtonCell to the typed stream *stream*. Returns **self**.

See also: – **read:**

Cell

Inherits From: Object

Declared In: appkit/Cell.h

Class Description

The Cell class provides a mechanism for displaying text or icons (that is, named NXImages) in a View without the overhead of a full View subclass. In particular, it provides much of the functionality of the Text class by providing access to a shared Text object used by all instances of Cell in an Application. Cells are also extremely useful for placing titles or icons at various locations in a custom subclass of View.

Cell is used heavily by the Control classes to implement their internal workings. Some subclasses of Control (notably Matrix) allow multiple Cells to be grouped and to act together in some cooperative manner. Thus, with a Matrix, a group of radio buttons can be implemented without needing a View for each button (and without needing a Text object for the text on each button).

The Cell class provides primitives for displaying text or an icon, editing text, formatting floating point numbers, maintaining state, highlighting, and tracking the mouse. It has several subclasses: SelectionCell, NXBrowserCell, and ActionCell (which in turn has the subclasses ButtonCell, SliderCell, TextFieldCell, and FormCell). Cell's **trackMouse:inRect:ofView:** method supports the target object and action method used to implement controls. However, Cell implements target/action features abstractly, deferring the details of implementation to subclasses of ActionCell.

The **initIconCell:** method is the designated initializer for Cells that display icons. The **initTextCell:** method is the designated initializer for Cells that display text. Override one or both of these methods if you implement a subclass of Cell that performs its own initialization. If you need to use target and action behavior, you may prefer to override ActionCell, which provides the default implementation of this behavior.

For more information on how Cell is used, see the Control class specification.

Instance Variables

```
char *contents;
id support;
struct _cFlags1 {
        unsigned int state:1;
        unsigned int highlighted:1;
        unsigned int disabled:1;
        unsigned int editable:1;
        unsigned int type:2;
        unsigned int freeText:1;
        unsigned int alignment:2;
        unsigned int bordered:1;
        unsigned int bezeled:1;
        unsigned int selectable:1;
        unsigned int scrollable:1;
        unsigned int entryType:3;
} cFlags1;
struct _cFlags2 {
        unsigned int continuous:1;
        unsigned int actOnMouseDown:1;
        unsigned int floatLeft:4;
        unsigned int floatRight:4;
        unsigned int autoRange:1;
        unsigned int actOnMouseDragged:1;
        unsigned int noWrap:1;
        unsigned int dontActOnMouseUp:1;
} cFlags2;
```

contents	The string for a text Cell; the image name for an icon Cell.
support	The Font for a text Cell; the NXImage for an icon Cell.
cFlags1.state	The state of the Cell (0 or 1).
cFlags1.highlighted	True if the Cell is highlighted.
cFlags1.disabled	True if the Cell is disabled.
cFlags1.editable	True if the text in the Cell is editable.
cFlags1.type	The type of the Cell.
cFlags1.freeText	True if the Cell should free contents when freeing the Cell.

cFlags1.alignment	The text alignment of the Cell.
cFlags1.bordered	True if the Cell has a solid border.
cFlags1.bezeled	True if the Cell has a bezeled border.
cFlags1.selectable	True if the text is selectable.
cFlags1.scrollable	True if the text is scrollable.
cFlags1.entryType	Data type accepted when the user types in a text Cell.
cFlags2.continuous	True if the Cell sends its action continuously to target while control is active.
cFlags2.actOnMouseDown	True if the Cell sends its action on a mouse-down.
cFlags2.floatLeft	Digits to left of decimal when text is floating-point number.
cFlags2.floatRight	Digits to right of decimal when text is floating-point number.
cFlags2.autoRange	True if the Cell autoranges decimal places when text is floating point number.
cFlags2.actOnMouseDragged	True if the Cell sends its action every time the mouse changes position.
cFlags2.noWrap	True if the Cell wraps text by character, false if by word.
cFlags2.dontActOnMouseUp	True if the Cell does *not* send its action on a mouse-up event.

Method Types

Initializing, copying, and freeing a Cell	– init – initIconCell: – initTextCell: – copyFromZone: – free
Determining component sizes	– calcCellSize: – calcCellSize:inRect: – calcDrawInfo: – getDrawRect: – getIconRect: – getTitleRect:

Setting the Cell's type	– setType: – type
Setting the Cell's state	– setState: – incrementState – state
Enabling and disabling the Cell	– setEnabled: – isEnabled
Setting the icon	– setIcon: – icon
Setting the Cell's value	– setDoubleValue: – doubleValue – setFloatValue: – floatValue – setIntValue: – intValue – setStringValue: – setStringValueNoCopy: – setStringValueNoCopy:shouldFree: – stringValue
Interacting with other Cells	– takeDoubleValueFrom: – takeFloatValueFrom: – takeIntValueFrom: – takeStringValueFrom:
Modifying text attributes	– setAlignment: – alignment – setFont: – font – setEditable: – isEditable – setSelectable: – isSelectable – setScrollable: – isScrollable – setTextAttributes: – setWrap:
Editing text	– edit:inView:editor:delegate:event: – endEditing: – select:inView:editor:delegate:start:length:

Validating input	– setEntryType: – entryType – isEntryAcceptable:
Formatting data	– setFloatingPointFormat:left:right:
Modifying graphic attributes	– setBezeled: – isBezeled – setBordered: – isBordered – isOpaque
Setting parameters	– setParameter:to: – getParameter:
Displaying	– controlView – drawInside:inView: – drawSelf:inView: – highlight:inView:lit: – isHighlighted
Target and action	– setAction: – action – setTarget: – target – setContinuous: – isContinuous – sendActionOn:
Assigning a tag	– setTag: – tag
Handling keyboard alternatives	– keyEquivalent
Tracking the mouse	+ prefersTrackingUntilMouseUp – mouseDownFlags – getPeriodicDelay:andInterval: – trackMouse:inRect:ofView: – startTrackingAt:inView: – continueTracking:at:inView: – stopTracking:at:inView:mouseIsUp:
Managing the cursor	– resetCursorRect:inView:
Archiving	– read: – write: – awake

Class Methods

prefersTrackingUntilMouseUp

+ (BOOL)**prefersTrackingUntilMouseUp**

Returns NO by default. Override this method to return YES if the Cell's View should allow it, after a mouse-down event, to track mouse-dragged and mouse-up events even if they occur outside the Cell's frame. For example, this method is overridden by SliderCell to ensure that a SliderCell in a Matrix doesn't stop responding to user input (and its neighbor start responding) just because its knob isn't dragged in a perfectly straight line.

See also: – **trackMouse:inRect:ofView:**

Instance Methods

action

– (SEL)**action**

Returns a null selector. This method is overridden by Action Cell and subclasses that actually implement a target object and action method.

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

alignment

– (int)**alignment**

Returns the alignment of text in the Cell. The return value can be one of three constants: NX_LEFTALIGNED, NX_CENTERED, or NX_RIGHTALIGNED.

See also: – **setAlignment:**

awake

– **awake**

Used during unarchiving to initialize static variables for the Cell class. Returns **self**.

See also: – **read:**

calcCellSize:

– **calcCellSize:**(NXSize *)*theSize*

Returns by reference the minimum width and height required for displaying the Cell. This method invokes **calcCellSize:inRect:** with the rectangle argument set to a rectangle with very large width and height. Override this method if that isn't the proper way to calculate the minimum width and height required for displaying the Cell. Returns **self**.

See also: – **calcCellSize:inRect:**

calcCellSize:inRect:

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

Returns by reference the minimum width and height required for displaying the Cell in the given rectangle. If it's not possible to fit, the width and/or height could be bigger than the ones of the provided rectangle. The computation is done by trying to size the Cell so that it fits in the rectangle argument (for example, by wrapping the text). If a choice must be made between extending the width or height of *aRect* to fit text, the height will be extended. Returns **self**.

See also: – **calcCellSize:**

calcDrawInfo:

– **calcDrawInfo:**(const NXRect *)*aRect*

Does nothing and returns **self**. Objects using Cells generally maintain a flag that informs them if any of their Cells has been modified in such a way that the location or size of the Cell should be recomputed. If so, **calcSize** is automatically invoked before displaying the Cell; that method invokes Cell's **calcDrawInfo:** for each Cell.

See also: – **calcSize** (Matrix)

continueTracking:at:inView:

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

Determines whether or not the Cell should keep tracking the mouse based on the positions provided. Returns YES if it can keep tracking, NO if should not. This method is invoked by **trackMouse:inRect:ofView:** as the mouse is dragged around inside the Cell. *lastPoint*

and *currentPoint* should be in *aView*'s coordinate system. By default, this method returns YES when the Cell is continuous (that is, when it should continually send action messages while the mouse is pressed or dragged). This method is often overridden to provide more sophisticated tracking behavior.

See also: – **trackMouse:inRect:ofView:**, – **startTrackingAt:inView:**, – **stopTracking:at:inView:mouseIsUp:**

controlView

– **controlView**

Returns **nil**. This method is implemented abstractly, since Cell doesn't record the View in which it's drawn. This method is overridden by ActionCell and its subclasses, which use the control View as the only argument in the action message when it's sent to the target.

See also: – **controlView** (ActionCell), – **drawSelf:inView:**, – **drawInside:inView:**

copyFromZone:

– **copyFromZone:**(NXZone *)*zone*

Allocates, initializes, and returns a copy of the receiving Cell. The copy is allocated from *zone* and is assigned the same contents as the receiver. When you subclass Cell, override this method to send the message

```
[super copyFromZone:zone];
```

then copy each of the subclass's unique instance variables separately in that same zone.

See also: – **copy** (Object)

doubleValue

– (double)**doubleValue**

Returns the receiving text Cell's value as a double-precision floating point number, by converting its string contents to a double using the standard C function **atof()**. Returns 0.0 if the Cell isn't a text Cell.

See also: – **setDoubleValue:**, – **floatValue**, – **intValue**, – **stringValue**, – **type**

drawInside:inView:

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

Draws the "inside" of the Cell. For the base Cell class, it's the same as **drawSelf:inView:** except that it doesn't draw the bezel or border if there is one. *cellFrame* should be the frame of the Cell (that is, the same as the *cellFrame* passed to **drawSelf:inView:**), *not* the rectangle returned by **getDrawRect:**. The PostScript focus must be locked on *aView* when this method is invoked. If the Cell's highlight flag is YES, then the Cell is highlighted (by swapping light gray and white throughout *cellFrame*; see the description of the Display PostScript operator **compositerect** for a description of highlighting). Returns **self**.

drawInside:inView: is usually invoked from the Control class's **drawCellInside:** method and is used to cause minimal drawing to be done in order to update the value displayed by the Cell when the **contents** is changed. This becomes more important in more complex Cells such as ButtonCell and SliderCell.

All subclasses of Cell which override **drawSelf:inView:** *must* override **drawInside:inView:**. **drawInside:inView:** should never invoke **drawSelf:inView:**, but **drawSelf:inView:** can—and often does—invoke **drawInside:inView:**.

See also: – **drawSelf:inView:**, – **lockFocus** (View), – **highlight:inView:lit:**, – **isHighlighted**, **compositerect** (Display PostScript operator)

drawSelf:inView:

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

Displays the contents of a Cell in a given rectangle of a given view. Your code must lock the focus on *aView* before invoking this method. It draws the border or bezel (if any), then invokes **drawInside:inView:**. A text Cell displays its text in the rectangle by using a global Text object. An icon Cell displays its icon centered in the rectangle if it fits in the rectangle, or by setting the icon origin on the rectangle origin if it doesn't fit. Nothing is displayed for a Cell of type NX_NULLCELL. Override this method if you want a display that is specific to your own subclass of Cell. Returns **self**.

See also: – **drawInside:inView:**, – **lockFocus** (View)

edit:inView:editor:delegate:event:

– **edit:**(const NXRect *)*aRect*
inView:*aView*
editor:*textObject*
delegate:*anObject*
event:(NXEvent *)*theEvent*

Begins editing of a Cell's text by using the Text object *textObject* in response to an NX_MOUSEDOWN event. *aRect* must be the one you have used when displaying the Cell. *theEvent* is the NX_MOUSEDOWN event. *anObject* is made the delegate of the Text object *textObject* used for the editing: it will receive messages such as **textDidEnd:endChar:**, **textWillEnd**, **textDidResize**, **textWillResize**, and others sent by the Text object while editing. If the receiver isn't a text Cell, no editing is performed, otherwise the Text object is sized to *aRect* and its superview is set to *aView*, so that it exactly covers the Cell. Then it's activated and editing begins. It's the responsibility of the delegate to end the editing, remove any data from *textObject* and invoke **endEditing:** on the Cell in the **textDidEnd:endChar:** method. Returns **self**.

See also: – **endEditing:**, Text class (Methods Implemented by the Delegate)

endEditing:

– **endEditing:***textObject*

Ends editing begun with **edit:inView:editor:delegate:event:** or **select:inView:editor:delegate:start:length:**. Usually this method is invoked by the **textDidEnd:endChar:** method of the object you are using as the delegate for the Text object (most often a Matrix or TextField). This method should remove the Text object from the view hierarchy and sets its delegate to **nil**. Returns **self**.

See also: – **edit:inView:editor:delegate:event:**, – **select:inView:editor:delegate:start:length:**, – **textDidEnd:endChar:** (Text class delegate method)

entryType

– (int)**entryType**

Returns the type of data allowed in the Cell. See **setEntryType:** for the list of valid types.

See also: – **setEntryType:**

floatValue

– (float)**floatValue**

Returns the receiving text Cell's value as a single-precision floating point number, by converting its string contents to a **double** using the C function **atof()** and then casting the result to a **float**. Returns 0.0 if the receiver isn't a text Cell.

See also: – **setFloatValue:**, – **doubleValue**, – **intValue**, – **stringValue**, – **type**

font

– **font**

Returns the Font used to display text in the Cell. Returns **nil** if the receiver isn't a text Cell.

See also: – **setFont:**, – **type**

free

– **free**

Frees the memory used by the Cell and returns **nil**. If the Cell's contents was set by copy (the default), then the contents is also freed.

getDrawRect:

– **getDrawRect:**(NXRect *)*theRect*

Given the bounds of the Cell in *theRect*, this method changes it to be the rectangle into which the Cell would draw its "insides" (everything but a bezel or border), and returns it by reference. In other words, this method calculates the rectangle which is touched by **drawInside:inView:**. However, your code should *not* use the rectangle returned by this method as the argument to **drawInside:inView:**. Returns **self**.

See also: – **getIconRect:**, – **getTitleRect:**, – **drawInside:inView:**

getIconRect:

– **getIconRect:**(NXRect *)*theRect*

Given the bounds of the Cell in *theRect*, this method changes it to be the rectangle into which the Cell would draw its icon, and returns it by reference. If the Cell doesn't draw an icon, *theRect* is untouched. Your code should *not* use the rectangle returned by this method as the argument to **drawInside:inView:**. Returns **self**.

See also: – **getDrawRect:**, – **getTitleRect:**, – **drawInside:inView:**

getParameter:

– (int)**getParameter:**(int)*aParameter*

Returns the value of one of the frequently accessed flags for a Cell. See **setParameter:to:** for a list of the parameters and corresponding methods. Since the parameters are also accessible through methods such as **isEnabled** and **isHighlighted**, you shouldn't need to use this method often.

See also: – **setParameter:to:**

getPeriodicDelay:andInterval:

– **getPeriodicDelay:**(float*)*delay* **andInterval:**(float*)*interval*

Returns by reference two values: the amount of time (in seconds) that a continuous button will pause before starting to periodically send action messages to the target object, and the interval (also in seconds) at which those messages are sent. Periodic messaging behavior is controlled by Cell's **sendActionOn:** and **setContinuous:** methods. (By default, Cell sends the action message only on mouse up events.) Override this method to return your own values. Returns **self**.

See also: – **setContinuous:**, – **sendActionOn:**

getTitleRect:

– **getTitleRect:**(NXRect *)*theRect*

Returns **self**, and, by reference in *theRect*, the rectangle into which the text will be drawn. If this Cell doesn't draw any text, *theRect* is untouched. Your code should *not* use the rectangle returned by this method as the argument to **drawInside:inView:**. Returns **self**.

See also: – **getDrawRect:**, – **getIconRect:**, – **drawInside:inView:**

highlight:inView:lit:

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

If the Cell's highlight status is different from *flag*, sets the Cell's highlight status to *flag* and, if *flag* is YES, highlights the rectangle *cellFrame* in *aView*. Your code must lock focus on *aView* before invoking this method. This method composites with NX_HIGHLIGHT inside the bounds of *cellFrame*. Override this method if you want more sophisticated highlighting behavior in a Cell subclass. Returns **self**.

Note that the highlighting that the base Cell class does will *not* appear when printed (although subclasses like TextFieldCell, SelectionCell, and ButtonCell can print themselves highlighted). This is because the base Cell class is transparent, and there is no concept of transparency in printed output.

See also: – **isHighlighted**, – **drawSelf:inView:**, – **drawInside:inView:**

icon

– (const char *)**icon**

Returns the name of the icon currently used by the Cell, if any, or NULL if the receiver isn't an icon Cell.

See also: – **setIcon:**, – **title**

incrementState

– **incrementState**

Adds 1 to the state of the Cell, wrapping around to 0 from the maximum value (which, for the Cell class, is 1). Returns **self**.

Subclasses may want to change the meaning of this method (to create multistate Cells, for example). Remember that if you want the visual appearance of the Cell to reflect a change in state, you must invoke **drawSelf:inView:** after altering the state. Your **drawSelf:inView:** implementation must draw the different states in different ways, since the default implementation doesn't do so.

See also: – **setState:**, – **drawSelf:inView:**

init

– **init**

Initializes and returns the receiver, a new Cell instance, as type NX_NULLCELL. This method is the designated initializer for cells without either text or an icon.

See also: – **initIconCell:**, – **initTextCell:**, – **setIcon:**, – **setText:**

initIconCell:

– **initIconCell:**(const char *)*iconName*

Initializes and returns the receiver, a new icon Cell instance (that is, its type is NX_ICONCELL). The icon is set to an NXImage with the name *iconName*. If *iconName* is NULL or an image for *iconName* is not found, the Cell will be initialized with a default icon, "NXsquare16". This method is the designated initializer for Cells that display an icon. If the Cell later has text assigned, its type will automatically change.

See also: – **icon**, – **setIcon:**, – **initTextCell:**, – **setText:**, – **init**, – **findImageFor:** (NXImage), – **name** (NXImage)

initTextCell:

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

Initializes and returns the receiver, a new text Cell instance, (that is, its type is NX_TEXTCELL). The string value is set to *aString*, or "Cell" if *aString* is NULL. This method is the designated initializer for text Cells.

See also: – **title**, – **setTitle:**, – **initIconCell:**, – **setIcon:**, – **init**

intValue

– (int)**intValue**

Returns the receiving text Cell's value as an integer, by converting its string contents to an **int** using the C function **atoi()**. Returns 0 if the receiver isn't a text Cell.

See also: – **setIntValue:**, – **doubleValue**, – **floatValue**, – **stringValue**, – **type:**

isBezeled

– (BOOL)**isBezeled**

Returns YES if the Cell draws itself with a bezeled border, NO otherwise. The default is NO.

See also: **– setBezeled:**, **– isBordered**

isBordered

– (BOOL)**isBordered**

Returns YES if the Cell draws itself surrounded by a 1-pixel black frame, NO otherwise. The default is NO.

See also: **– setBordered:**, **– isBezeled**

isContinuous

– (BOOL)**isContinuous**

Returns YES if the Cell continuously sends its action message to the target object when tracking. This usually has meaning only for subclasses of Cell that implement instance variables and methods for target/action functionality, such as ActionCell; certain Control subclasses, specifically Matrix, send a default action to a default target even if the Cell doesn't have a target and action.

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

isEditable

– (BOOL)**isEditable**

Returns YES if text in the Cell is editable (and therefore also selectable), NO otherwise. The default is NO.

See also: **– setEditable:**, **– isSelectable**

isEnabled

– (BOOL)**isEnabled**

Returns YES if the Cell is enabled, NO otherwise. The default is YES. A Cell's enabled status is used primarily in event handling and display: It affects the behavior of methods for mouse tracking and text editing, by allowing or disallowing changes to the Cell within those methods, and only allows the Cell to highlight or set a cursor rectangle if it's enabled. You can still affect many Cell attributes programmatically (**setState:**, for example, will still work).

See also: – **setEnabled:**, – **trackMouse:inRect:ofView:**

isEntryAcceptable:

– (BOOL)**isEntryAcceptable:**(const char *)*aString*

Tests whether *aString* matches the Cell's entry type, as set by the **setEntryType:** method. Returns YES if *aString* is acceptable by the receiving Cell, NO otherwise. For example, a text Cell of type NX_INTTYPE accepts strings that represent integers, but not floating point numbers or words. If *aString* is NULL or empty, this method returns YES.

This method is invoked by Form, Matrix, and other Controls to see if a new text string is acceptable for this Cell. This method doesn't check for overflow. It can be overridden to enforce specific restrictions on what the user can type into the Cell.

See also: – **setEntryType:**

isHighlighted

– (BOOL)**isHighlighted**

Returns YES if the Cell is highlighted, NO otherwise.

See also: – **highlight:inView:lit:**

isOpaque

– (BOOL)**isOpaque**

Returns YES if the Cell is opaque (that is, if it draws over every pixel in its frame), NO otherwise. The base Cell class is opaque if and only if it draws a bezel. Subclasses that draw differently should override this based on how they perform their drawing.

See also: – **setBezeled:**

isScrollable

– (BOOL)**isScrollable**

Returns YES if typing past an end of the text in the Cell will cause the Cell to scroll to follow the typing. The default return value is NO.

See also: – **setScrollable:**

isSelectable

– (BOOL)**isSelectable**

Returns YES if the text in the Cell is selectable, NO otherwise. The default is NO.

See also: – **setSelectable:**, – **isEditable**

keyEquivalent

– (unsigned short)**keyEquivalent**

Returns 0, as Cell provides no support for key equivalents. Subclasses can implement key equivalents, and can override this method to return the key equivalent for the receiver.

See also: – **setKeyEquivalent:** (ButtonCell), – **keyEquivalent** (ButtonCell)

mouseDownFlags

– (int)**mouseDownFlags**

Returns the flags (for example, NX_SHIFTMASK) that were set when the mouse went down to start the current tracking session. This method is only valid during tracking. It doesn't work if the target of the Cell initiates another Cell tracking loop as part of its action method (as does PopUpList).

See also: – **sendActionOn:**

read:

– **read:**(NXTypedStream *)*stream*

Reads the Cell from the typed stream *stream*.

See also: – **write:**, – **awake**

resetCursorRect:inView:

– **resetCursorRect:**(const NXRect *)*cellFrame* **inView:***aView*

If the receiver is a textCell, then a cursor rectangle is added to *aView* (with **addCursorRect:cursor:**). This allows the cursor to change to an I-beam when it passes over the Cell. Override this method to change the cursor for an icon Cell, or to provide a different cursor for a text Cell.

See also: – **addCursorRect:cursor:** (View, Control)

select:inView:editor:delegate:start:length:

– **select:**(const NXRect *)*aRect*
inView:*aView*
editor:*aTextObject*
delegate:*anObject*
start:(int)*selStart*
length:(int)*selLength*

Uses *aTextObj* to select text in the Cell identified by *selStart* and *selLength*, which will be highlighted and selected as though the user had dragged the cursor over it. This method is similar to **edit:inView:editor:delegate:event:**, except that it can be invoked in any situation, not only on a mouse-down event.

See also: – **edit:inView:editor:delegate:event:**

sendActionOn:

– (int)**sendActionOn:**(int)*mask*

Resets flags to determine when the action is sent to the target while tracking. Can be any logical combination of:

NX_MOUSEUPMASK
NX_MOUSEDOWNMASK
NX_MOUSEDRAGGEDMASK
NX_PERIODICMASK

The default is NX_MOUSEUPMASK. You can also use the **setContinuous:** method to turn on the flag corresponding to NX_PERIODICMASK (**cflags2.continuous**) or

NX_MOUSEDRAGGEDMASK (**cflags2.actOnMouseDragged**), whichever is appropriate to the given subclass of Cell.

This method returns an event mask built from the old flags.

See also: – **setContinous:**

setAction:

– **setAction:**(SEL)*aSelector*

Does nothing. This method is overridden by Action Cell and its subclasses, which actually implement the target object and action method. It is also overriden by NXBrowserCell to provide access to its NXBrowser's action method. Returns **self**.

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

setAlignment:

– **setAlignment:**(int)*mode*

Sets the alignment of text in the Cell. *mode* should be one of three constants: NX_LEFTALIGNED, NX_CENTERED, or NX_RIGHTALIGNED. Returns **self**.

See also: – **alignment**, – **setWrap:**

setBezeled:

– **setBezeled:**(BOOL)*flag*

If *flag* is YES, then the Cell draws itself surrounded by a bezel; if NO, it doesn't. **setBordered:** and **setBezeled:** are mutually exclusive. Returns **self**.

See also: – **isBezeled**, – **setBordered:**

setBordered:

– **setBordered:**(BOOL)*flag*

If *flag* is YES, then the Cell draws itself surrounded by a 1-pixel black frame; if NO, it doesn't. **setBordered:** and **setBezeled:** are mutually exclusive. Returns **self**.

See also: – **isBordered**, – **setBezeled:**

setContinuous:

– **setContinuous:**(BOOL)*flag*

Sets whether a Cell continuously sends its action message to the target object when tracking. Normally, this method will set the continuous (**cflags2.continuous**) or the mouse-dragged flag (**cflags2.actOnMouseDragged**), depending on which setting is appropriate to the subclass implementing it. In the base Cell class, this method sets the continuous flag. These settings usually have meaning only for ActionCell and its subclasses which implement the instance variables and methods that provide target/action functionality. Some Control subclasses, specifically Matrix, send a default action to a default target when a Cell doesn't provide a target or action.

See also: – **isContinuous**, – **sendActionOn:**

setDoubleValue:

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

Sets the contents of the Cell to the string value representing the double-precision floating point number *aDouble*, ignoring the entry type of the Cell. Does nothing if the receiver isn't a text Cell. Returns **self**.

See also: – **doubleValue**, – **setFloatValue:**, – **setIntValue:**, – **setStringValue:**, – **entryType**, – **type**

setEditable:

– **setEditable:**(BOOL)*flag*

If *flag* is YES, then the text is made both editable and selectable. If *flag* is NO, and the text was not selectable before editing was last enabled (that is, before this message was last sent with an argument of YES), then the text is returned to not being selectable. Returns **self**.

See also: – **isEditable**, – **setSelectable:**, – **edit:inView:editor:delegate:event:**

setEnabled:

– **setEnabled:**(BOOL)*flag*

Sets the enabled status of the Cell. A Cell's enabled status is used primarily in event handling and display: It affects the behavior of methods for mouse tracking and text editing, by allowing or disallowing changes to the Cell within those methods, and only

allows the Cell to highlight or set a cursor rectangle if it's enabled. Many Cell attributes can still be altered programmatically (**setState:**, for example, will still work). Returns **self**.

See also: – **isEnabled**

setEntryType:

– **setEntryType:**(int)*aType*

This method sets the data format allowed in the Cell. *aType* is one of these seven constants, allowing only the corresponding numeric string values to be entered:

NX_ANYTYPE	No restrictions
NX_INTTYPE	Integer values
NX_FLOATTYPE	Single-precision floating point values
NX_DOUBLETYPE	Double-precision floating point values
NX_POSINTTYPE	Positive integer values
NX_POSFLOATTYPE	Positive single-precision floating point values
NX_POSDOUBLETYPE	Positive double-precision floating point values

If the receiver isn't a text Cell, it's converted to type NX_TEXTCELL, in which case its font is set to the user's system font at 12.0 point, and its string value is set to "Cell" (even for text Cells that display numbers).

The entry type is checked by the **isEntryAcceptable:** method. That method is used by Controls that contain editable text (such as Matrix and TextField) to validate that what the user has typed is correct. If you want to have a custom Cell accept some specific type of data (other than those listed above), override the **isEntryAcceptable:** method to check for the validity of the data the user has entered.

See also: – **entryType**, – **isEntryAcceptable:**, – **setFloatingPointFormat:left:right:**

setFloatingPointFormat:left:right:

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

Sets whether floating-point numbers are autoranged, and sets the sizes of the fields to the left and right of the decimal point. *leftDigits* specifies the maximum number of digits to the left of the decimal point, and *rightDigits* specifies the number of digits to the right (the fractional digit places will be padded with zeros to fill this width). However, if a number is too large to fit its integer part in *leftDigits* digits, as many places as are needed on the left are effectively removed from *rightDigits* when the number is displayed.

If *autoRange* is YES, *leftDigits* and *rightDigits* are simply added to form a maximum total field width for the Cell (plus 1 for the decimal point). The fractional part will be padded with zeros on the right to fill this width, or truncated as much as possible (up to removing the decimal point and displaying the number as an integer). The integer portion of a number is never truncated—that is, it is displayed in full no matter what the field width limit is.

leftDigits must be between 0 and 10. *rightDigits* must be between 0 and 14. If *leftDigits* is 0, then the default **printf()** formatting applies. If *rightDigits* is 0, then the decimal and the fractional part of the floating-point number are truncated (that is, the floating-point number is printed as if it were an integer). If the entry type of the Cell isn't already NX_FLOATTYPE, NX_POSFLOATTYPE, NX_DOUBLETYPE, or NX_POSDOUBLETYPE, it's set to NX_FLOATTYPE. Returns **self**.

See also: – **setEntryType:**

setFloatValue:

– **setFloatValue:**(float)*aFloat*

Sets the contents of the Cell to the string value representing the single-precision floating point number *aFloat*, ignoring the entry type of the Cell. Does nothing if the receiver isn't a text Cell. Returns **self**.

See also: – **floatValue**, – **setDoubleValue:**, – **setIntValue:**, – **setStringValue:**, – **entryType**, – **type**

setFont:

– **setFont:***fontObject*

Sets the Font to be used when displaying text in the Cell. Does nothing if the receiver isn't a text Cell. Returns **self**.

See also: – **font**

setIcon:

– **setIcon:**(const char *)*iconName*

Sets the Cell's icon to *iconName* (an NXImage object with that name). *iconName* is stored as the Cell's contents, and the NXImage is stored as its support. If the Cell isn't an icon cell, it's converted; if the Cell was a text Cell, the text string is freed if necessary. If *iconName* is NULL or an empty string, or if an image can't be found for *iconName*, the Cell has its icon set to the standard system bitmap "NXsquare16".

If you specify a name for which an image can't be found, no change is made. Your code can verify that the icon was properly changed by comparing the values returned by the **type** or **icon** methods before and after invoking **setIcon:**. Returns **self**.

See also: – **icon**, – **findImageNamed** (NXImage), – **initIconCell:**

setIntValue:

– **setIntValue:**(int)*anInt*

Sets the contents of the Cell to the string value representing the integer *anInt*. Does nothing if the receiver isn't a text Cell. This method ignores the entry type of the Cell. Returns **self**.

See also: – **intValue**, – **setDoubleValue:**, – **setFloatValue:**, – **setStringValue:**, – **type**, – **entryType**

setParameter:to:

– **setParameter:**(int)*aParameter* **to:**(int)*value*

Sets the value of one of the Cell's parameters to *value*, and returns **self**. You don't normally use this method, since these parameters can be set using specific methods such as **setEditable:**. In this method, the parameters is identified by *aParameter*, a symbolic constant defined in the header file **appkit/Cell.h**. The following table lists these constants with the corresponding methods for setting and getting the value of the related parameters:

Parameter Constant	Equivalent Methods
NX_CELLDISABLED	setEnabled:, isEnabled
NX_CELLHIGHLIGHTED	highlightInView:lit:, isHighlighted
NX_CELLSTATE	setState:, incrementState, state
NX_CELLEDITABLE	setEditable:, isEditable

Use of this method is discouraged as it could produce unpredictable results in subclasses. It's much safer to invoke the appropriate parameters-specific method.

See also: – **getParameter:**

setScrollable:

– **setScrollable:**(BOOL)*flag*

Sets whether the Cell will scroll to follow typing while being edited. Returns **self**.

See also: – **isScrollable**, – **edit:inView:editor:delegate:event:**

setSelectable:

– **setSelectable:**(BOOL)*flag*

If *flag* is YES, then the text is made selectable but not editable. If NO, then the text is static (neither editable nor selectable). To make text in a Cell both selectable and editable, send it a **setEditable:** message. Returns **self**.

See also: – **isSelectable:**, – **isEditable**, – **edit:inView:editor:delegate:event:**

setState:

– **setState:**(int)*value*

Sets the state of the Cell to 0 if *value* is 0, to 1 otherwise. Returns **self**.

See also: – **state**, – **incrementState**

setStringValue:

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

Copies *aString* as the receiver's contents. If the receiver isn't a text Cell, this method converts it to that type, setting its font to the user's system font at 12 points. Returns **self**.

If the receiver was an icon Cell, the NXImage for that icon is *not* freed; your code should retrieve it beforehand and free it after sending this message.

If floating point formatting has been set (with **setFloatingPointParameters:left:right:**) and the entry type of the Cell is a floating point number type, then the string is tested to determine whether it represents a floating point number; if so, the string is displayed according to that floating point format.

See also: – **setStringValueNoCopy:**, – **setStringValueNoCopy:shouldFree:**, – **stringValue**, – **setDoubleValue:**, – **setFloatValue:**, – **setIntValue:**, – **setFloatingPointFormat:left:right:**

setStringValueNoCopy:

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

Similar to **setStringValue:** but doesn't make a copy of *aString*. The Cell records that it doesn't have to dispose of its contents when it receives a **free** message. Note that if a string

is set this way, floating-point formatting can't be applied (since a shared string can't be altered). Returns **self**.

See also: – **setStringValue:**, – **setStringValueNoCopy:shouldFree:**, – **stringValue**

setStringValueNoCopy:shouldFree:

– **setStringValueNoCopy:**(char *)*aString* **shouldFree:**(BOOL)*flag*

Similar to **setStringValueNoCopy:**, but the sender can specify in *flag* if the contents should be freed when the Cell receives a **free** message. Note that if a string is set this way, floating-point formatting isn't applied. Returns **self**.

If the contents was already the same string as *aString* (the same pointer, not the same string value), the free-contents flag can't be set set to YES. That is, you can't set a string as non-freeable and later change it to be freeable by reinvoking this method with that same string; you can, however, change it from freeable to nonfreeable.

See also: – **setStringValue:**, – **setStringValueNoCopy:**, – **stringValue**

setTag:

– **setTag:**(int)*anInt*

Does nothing. This method is overridden by ActionCell and its subclasses to support Controls with multiple Cells (Matrix and Form). Override this method to provide a way to identify Cells. Returns **self**.

See also: – **tag**, – **findCellWithTag:** (Matrix, Menu classes)

setTarget:

– **setTarget:***anObject*

Does nothing. This method is one of several overridden by ActionCell and subclasses to implement target/action functionality. Returns **self**.

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

setTextAttributes:

– **setTextAttributes:***textObject*

Invoked just before any drawing or editing occurs in the Cell. This method is intended to be overridden. If you do override this method you must include this line first:

```
[super setTextAttributes:textObject];
```

If you don't, you risk inheriting drawing attributes from the last Cell which drew any text. You should invoke only the **setBackgroundGray:** and **setTextGray:** Text instance methods. Don't set any other parameters in the Text object.

This method normally returns *textObject*. If you want to substitute some other Text object to draw with (but not edit, since editing always uses the window's field editor), you can return that object instead of *textObject* and it will be used for the draw that caused **setTextAttributes:** to be invoked.

TextFieldCell, a subclass of ActionCell, allows you to set the grays without creating your own subclass of Cell. You only need to subclass Cell to control the gray values if you don't want all the functionality (and instance variable usage) of an ActionCell.

The default values for text attributes are as follows. If the Cell is enabled, its text gray will be NX_BLACK, otherwise it will be NX_DKGRAY. If the Cell has a bezel, then its background gray will be NX_WHITE, otherwise it will be NX_LTGRAY. The Text object does *not* paint the background gray before drawing; it only uses the background gray to erase characters while editing. The Cell class does paint the NX_WHITE background when it draws a bezeled Cell, but doesn't paint any background otherwise (that is, it's transparent).

Note that most of the other text object attributes can be set with Cell methods (**setFont:**, **setAlignment:**, **setWrap:**) so you need only override this method if you need to set the gray values. Returns **self**.

setType:

– **setType:**(int)*aType*

Sets the type of the Cell. *aType* should be NX_TEXTCELL, NX_ICONCELL, or NX_NULLCELL. If *aType* is NX_TEXTCELL and the receiver isn't currently a text Cell, then the font is set to the user's system font in 12.0 point; its string value is set to "Cell". If *aType* is NX_ICONCELL and the receiver isn't an icon Cell, then the icon set to the default, "NXsquare16".

See also: – **type**, – **init**, – **initIconCell:**, – **initTextCell:**, – **setIcon:**, – **setText:**

setWrap:

– **setWrap:**(BOOL)*flag*

If *flag* is YES, text will be wrapped to word breaks. If *flag* is NO, it will be truncated. The default is YES. This setting has effect only when displaying text, not when editing, and only applies to Cells whose alignment is NX_LEFTALIGNED (centered and right-aligned text always wraps to word breaks).

See also: – **setAlignment:**

startTrackingAt:inView:

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

This method is invoked from **trackMouse:inRect:ofView:** the first time the mouse appears in the Cell needing to be tracked. Override to provide implementation-specific tracking behavior. This method should return YES if it's OK to track based on this starting point, and *only* if the Cell is continuous; otherwise it should return NO.

See also: – **trackMouse:inRect:ofView:**, – **continueTracking:at:inView:**, – **stopTracking:at:inView:mouseIsUp:**, – **isContinuous**, – **mouseDownFlags**

state

– (int)**state**

Returns the state of the Cell (0 or 1). The default is 0.

See also: – **setState:**, – **incrementState**

stopTracking:at:inView:mouseIsUp:

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

Invoked from **trackMouse:inRect:ofView:** when the mouse has left the bounds of the Cell, or the mouse button has gone up. *flag* is YES if the mouse button went up to cause this method to be invoked. The default behavior is to do nothing and return **self**. This method is often overridden to provide more sophisticated tracking behavior.

See also: – **trackMouse:inRect:ofView:**, – **startTrackingAt:inView:**, – **continueTracking:at:inView:**

stringValue

– (const char *)**stringValue**

Returns the contents of the Cell as a string.

See also: – **setStringValue:**, – **doubleValue**, – **floatValue**, – **intValue**

tag

– (int)**tag**

Returns –1. This method is overridden by ActionCell and its subclasses to support multiple-Cell controls (Matrix and Form). Override this method if you want to use tags to identifiy Cells. Returns **self**.

See also: – **setTag:**, – **findCellWithTag:** (Matrix, Menu classes)

takeDoubleValueFrom:

– **takeDoubleValueFrom:***sender*

Sets the Cell's double-precision floating point value to the value returned by *sender*'s **doubleValue** method. *sender* must be of a class that implements the **doubleValue** method. Returns **self**.

This method can be used in action messages between Cells. It permits one Cell (the sender) to affect the value of another Cell (the receiver). For example, a TextFieldCell can be made the target of a SliderCell, which will send it a **takeDoubleValueFrom:** action message. The TextFieldCell will get the return value of the SliderCell's **doubleValue** method, turn it into a text string, and display it.

See also: – **takeDoubleValueFrom:** (Control), – **setDoubleValue:**

takeFloatValueFrom:

– **takeFloatValueFrom:***sender*

Sets the Cell's single-precision floating-point value to the value returned by *sender*'s **floatValue** method. *sender* must be of a class that implements the **floatValue** method. Returns **self**.

This method is similar to **takeDoubleValueFrom:** except it works with floats rather than doubles.

See also: – **takeFloatValueFrom:** (Control), – **setFloatValue:**

takeIntValueFrom:

– **takeIntValueFrom:***sender*

Sets the Cell's integer value to the value returned by *sender*'s **intValue** method. *sender* must be of a class that implements the **intValue** method. Returns **self**.

This method is similar to **takeDoubleValueFrom:** except it works with ints rather than doubles.

See also: – **takeIntValueFrom:** (Control), – **setIntValue:**

takeStringValueFrom:

– **takeStringValueFrom:***sender*

Sets the Cell's string value to the value returned by *sender*'s **stringValue** method. *sender* must be of a class that implements the **stringValue** method. Returns **self**.

This method is similar to **takeDoubleValueFrom:** except it works with strings rather than doubles.

See also: – **takeStringValueFrom:** (Control), – **setStringValue:**

target

– **target**

Returns **nil**. This method is one of those overridden by ActionCell and subclasses to implement target/action functionality.

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

trackMouse:inRect:ofView:

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

Invoked by a Control to initiate the tracking behavior of a Cell. It's generally not overridden since the default implementation invokes other Cell methods that can be overridden to handle specific events in a dragging session. Returns YES if the mouse goes up in *cellFrame*, NO otherwise.

This method first invokes **startTrackingAt:inView:**. If that method returns YES, then as mouse-dragged events are intercepted, **continueTracking:at:inView:** is invoked, and, finally, when the mouse leaves the bounds or if the mouse button goes up, **stopTracking:at:inView:mouseIsUp:** is invoked (if *cellFrame* is NULL, then the bounds are considered infinitely large). You usually override one or more of these methods to respond to specific mouse events.

If the other tracking methods are insufficient for your needs, override this method directly. It's this method's responsibility to invoke *aView*'s **sendAction:to:** method when appropriate (before, during, or after tracking) and to return YES if and only if the mouse goes up within the Cell during tracking. If the Cell's action is sent on a mouse down event, then **startTrackingAt:inView:** is invoked *before* the action is sent and the mouse is tracked until it goes up or out of bounds. If the Cell sends its action periodically, then the action is sent periodically to the target even if the mouse isn't moving (although **continueTracking:at:inView:** is only invoked when the mouse changes position). If the Cell's action is sent on a mouse dragged event, then **continueTracking:at:inView:** is invoked *before* the action is sent. The state of the Cell is incremented (with **incrementState**) before the action is sent and after **stopTracking:at:inView:** is invoked when the mouse goes up.

See also: – **startTrackingAt:inView:**, – **continueTracking:at:inView:**, – **stopTracking:at::inView:mouseIsUp:**

type

– (int)**type**

Returns the type of the Cell, which can be either NX_NULLCELL, NX_ICONCELL or NX_TEXTCELL.

See also: – **setType:**

write:

– **write:**(NXTypedStream *)*stream*

Writes the Cell to the typed stream *stream*. Returns **self**.

See also: – **read:**

ClipView

Inherits From: View : Responder : Object

Declared In: appkit/ClipView.h

Class Description

A ClipView object lets you scroll a document that may be larger than the ClipView's frame rectangle, clipping the visible portion of the document to the frame. The document, which must be a View object, is called the ClipView's *document view.* A ClipView's document view, which is set through the **setDocView:** method, is the ClipView's only subview. You can set the cursor that's displayed when the mouse enters a ClipView's frame (in other words, when it's poised over the document view) through the **setDocCursor:** method.

When the ClipView is instructed to scroll its document view, it copies as much of the previously visible document as possible, unless it received a **setCopyOnScroll:NO** message. The ClipView then sends its document view a message to either display or mark as invalid the newly exposed region(s) of the ClipView. By default it will invoke the document view's **display::** method, but if the ClipView received a **setDisplayOnScroll:NO** message, it will invoke the document view's **invalidate::** method.

The ClipView sends its superview (usually a ScrollView) a **reflectScroll:** message whenever the relationship between the ClipView and the document view has changed. This allows the superview to update itself to reflect the change—for example, the ScrollView class uses this method to change the position of its scrollers when the user causes the document to autoscroll.

You don't normally use the ClipView class directly; it's provided primarily as the scrolling machinery for the ScrollView class. However, you might use the ClipView class to implement a class similar to ScrollView.

Instance Variables

float **backgroundGray**;
id **docView**;
id **cursor**;

backgroundGray	The gray value used to fill the ClipView's background.
docView	The ClipView's document view.
cursor	The cursor that's used within the ClipView's frame.

Method Types

Initializing the class	+ initialize
Initializing and freeing a ClipView	– initFrame: – free
Modifying the frame rectangle	– moveTo:: – rotateTo: – sizeTo::
Modifying the coordinate system	– rotate: – scale:: – setDrawOrigin:: – setDrawRotation: – setDrawSize:: – translate::
Managing component Views	– docView – setDocView: – getDocRect: – getDocVisibleRect: – resetCursorRects – setDocCursor:
Modifying graphic attributes and displaying	– backgroundGray – setBackgroundGray: – backgroundColor – setBackgroundColor: – drawSelf::

Scrolling	– autoscroll: – constrainScroll: – rawScroll: – setCopyOnScroll: – setDisplayOnScroll:
Coordinating with other Views	– descendantFlipped: – descendantFrameChanged:
Archiving	– awake – read: – write:

Class Methods

initialize

+ initialize

Sets the current version of the ClipView class. You never invoke this method directly; it's sent for you when the application starts. Returns **self**.

Instance Methods

autoscroll:

– **autoscroll:**(NXEvent *)*theEvent*

Performs automatic scrolling of the document. You never invoke this method directly; instead, the ClipView's document view should send **autoscroll:** to itself while inside a modal event loop initiated by a mouse-down event when the mouse is dragged outside the ClipView's frame. The View class implements **autoscroll:** to forward the message to the View's superview; thus is the message forwarded to the ClipView.

Returns **nil** if no scrolling occurs; otherwise returns **self**.

See also: – **autoscroll:** (View)

awake

– awake

You never invoke this method directly; it's invoked automatically after the ClipView has been read from an archive file. Returns **self**.

backgroundColor

– (NXColor)**backgroundColor**

Returns the color of the ClipView's background. If the background gray value has been set but no color has been set, the color equivalent of the background gray value is returned. If neither value has been set, the background color of the ClipView's window is returned.

See also: – **backgroundGray**, – **setBackgroundColor:**, – **setBackgroundGray:**, – **backgroundColor** (Window), **NXConvertGrayToColor()**

backgroundGray

– (float)**backgroundGray**

Returns the gray value of the ClipView's background. If no value has been set, the gray value of the ClipView's window is returned.

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

constrainScroll:

– **constrainScroll:**(NXPoint *)*newOrigin*

Ensures that the document view is not scrolled to an undesirable position. This method is invoked by the private method that all scrolling messages go through before it invokes **rawScroll:** or **scrollClip:to:**. The default implementation keeps as much of the document view visible as possible. You may want to override this method to provide alternate constraining behavior. *newOrigin* is the desired new origin of the ClipView's bounds rectangle, given in ClipView's coordinate system. Returns **self**.

See also: – **rawScroll:**

descendantFlipped:

– **descendantFlipped:***sender*

Changes the ClipView's coordinate system orientation (unflipped or flipped) to match that of the document view. You never invoke this method directly; it's invoked automatically when the document view's orientation changes. Returns **self**.

descendantFrameChanged:

– **descendantFrameChanged:***sender*

Notifies the ClipView that its document view has been resized or moved. The ClipView may then scroll and/or redisplay the document view, and the ClipView may also notify its superview to reflect the changes in the scroll position. You never invoke this method directly, nor should you override it in a subclass. Returns **self**.

See also: – **moveTo::** (View), – **sizeTo::** (View), – **reflectScroll:** (ScrollView), – **notifyAncestorWhenFrameChanged:** (View), – **setDocView:**

docView

– **docView**

Returns the ClipView's document view.

See also: – **setDocView:**

drawSelf::

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

Overrides View's **drawSelf::** method to fill the portions of the ClipView that aren't covered by opaque portions of the document view. If a color value has been set and the ClipView is drawing itself on a color screen, the ClipView draws its background with the color value, otherwise it draws its background using its gray value. Returns **self**.

See also: – **backgroundColor:**, – **backgroundGray:**, – **drawSelf::** (View)

free

– **free**

Frees the ClipView and its subviews.

getDocRect:

– **getDocRect:**(NXRect *)*aRect*

Returns, by reference in *aRect*, the smallest rectangle that encloses both the document view's frame and the ClipView's frame. The origin of the rectangle is always set to that of the document view's frame.

The document rectangle is used in conjunction with the ClipView's bounds rectangle to determine values for any indicators of relative position and size between the ClipView and the document view. The ScrollView uses these rectangles to set the size and position of the Scrollers' knobs. Returns **self**.

See also: – **reflectScroll:** (ScrollView)

getDocVisibleRect:

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

Returns, by reference in *aRect*, the portion of the document view that's visible within the ClipView. The visible rectangle is given in the document view's coordinate system. Note that this rectangle doesn't reflect the effects of any clipping that may occur above the ClipView itself. To get the portion of the document view that's guaranteed to be visible, send it a **getVisibleRect:** message. Returns **self**.

See also: – **getVisibleRect:** (View)

initFrame:

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

Initializes the ClipView, which must be a newly allocated ClipView instance. The ClipView's frame rectangle is made equivalent to that pointed to by *frameRect*. This method is the designated initializer for the ClipView class, and can be used to initialize a ClipView allocated from your own zone. By default, clipping is enabled, and the ClipView is set to opaque. A ClipView is initialized without a document view. Returns **self**.

moveTo::

– **moveTo:**(NXCoord)*x* **:**(NXCoord)*y*

Moves the origin of the ClipView's frame rectangle to (*x*, *y*) in its superview's coordinates. Returns **self**.

rawScroll:

– **rawScroll:**(const NXPoint *)*newOrigin*

Performs scrolling of the document view. This method sets the ClipView's bounds rectangle origin to *newOrigin*. Then it copies as much of the previously visible document as possible, unless it received a **setCopyOnScroll:**NO message. It then sends its document view a message to either display or invalidate the newly exposed region(s) of the ClipView. By default it will invoke the document view's **display::** method, but if the ClipView received a **setDisplayOnScroll:**NO message, it will invoke the document view's **invalidate::** method. The **rawScroll:** method doesn't send a **reflectScroll:** message to its superview; that message is sent by the method that invokes **rawScroll:**. Note also that while the ClipView provides clipping to its frame, it doesn't clip to the update rectangles.

This method is used by a private method through which all scrolling passes, and is invoked if the ClipView's superview does not implement the **scrollClip:to:** method. If the ClipView's superview does implement **scrollClip:to:**, that method should invoke **rawScroll:**. This mechanism is provided so that the ClipView's superview can coordinate scrolling of multiple tiled ClipViews. (Note that ScrollView doesn't implement the **scrollClip:to:** method.) Returns **self**.

read:

– **read:**(NXTypedStream *)*stream*

Reads the ClipView and its document view from the typed stream *stream*. Returns **self**.

See also: – **write:**

resetCursorRects

– **resetCursorRects**

Resets the cursor rectangle for the document view to the bounds of the ClipView. Returns **self**.

See also: – **setDocCursor:**, – **addCursorRect:cursor:** (View)

rotate:

– **rotate:**(NXCoord)*angle*

Disables rotation of the ClipView's coordinate system. You also should not rotate the ClipView's document view, nor should you install a ClipView as a subview of a rotated view. The proper way to rotate objects in the document view is to perform the rotation in your document view's **drawSelf::** method. Returns **self**.

rotateTo:

– **rotateTo:**(NXCoord)*angle*

Disables rotation of the ClipView's frame rectangle. This method also disables ClipView's inherited **rotateBy:** method. Returns **self**.

See also: – **rotate:**

scale::

– **scale:**(NXCoord)*x* **:**(NXCoord)*y*

Rescales the ClipView's coordinate system by a factor of *x* for its x-axis, and by a factor of *y* for its y-axis. Since the document view's coordinate system is measured relative to the ClipView's coordinate system, the document view is redisplayed and a **reflectScroll:** message may be sent to the ClipView's superview. Returns **self**.

See also: – **reflectScroll:** (ScrollView)

setBackgroundColor:

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

Sets the color of the ClipView's background. This color is used to fill the area inside the ClipView that's not covered by opaque portions of the document view. If no background gray has been set for the ClipView, this method sets it to the gray component of the color. Returns **self**.

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

setBackgroundGray:

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

Sets the gray value of the ClipView's background. This gray is used to fill the area inside the ClipView that's not covered by opaque portions of the document view. *value* must lie in the range from 0.0 (black) to 1.0 (white). Returns **self**.

See also: – **backgroundColor**, – **backgroundGray**, – **setBackgroundColor**

setCopyOnScroll:

– **setCopyOnScroll:**(BOOL)*flag*

Determines whether visible portions of the document view will be copied when scrolling occurs. If *flag* is YES, scrolling will copy as much of the document as possible to scroll the View, allowing the document view to update only the newly exposed portions of itself. If *flag* is NO, the document view is responsible for redrawing its entire visible portion. This should only rarely be changed from the default value (YES). Returns **self**.

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 isn't displayed. This should only rarely be changed from the default setting of YES. Returns **self**.

See also: – **rawScroll:**, – **display::** (View), – **invalidate::** (View)

setDocCursor:

– **setDocCursor:***anObj*

Sets the cursor to be used inside the ClipView's bounds. *anObj* should be an NXCursor object. Returns the old cursor.

setDocView:

– **setDocView:***aView*

Sets *aView* as the ClipView's document view. A ClipView can have only one document view; invoking this method removes the previous document view, if any. This method initializes the document view with **notifyAncestorWhenFrameChanged:YES** and **notifyWhenFlipped:YES** messages. The origin of the document view's frame is initially set to be coincident with the origin of the ClipView's bounds. If the ClipView is contained within a ScrollView, you should send the ScrollView the **setDocView:** message and have the ScrollView pass this message on to the ClipView. Returns the old document view, or **nil** if there was none.

See also: – **setDocView:** (ScrollView)

setDrawOrigin::

– **setDrawOrigin:**(NXCoord)*x* **:**(NXCoord)*y*

Overrides the View method so that changes in the ClipView's coordinate system are reflected in the displayed document view. This method may redisplay the document view, and a **reflectScroll:** message may be sent to the ClipView's superview. Returns **self**.

See also: – **setDrawOrigin::** (View)

setDrawRotation:

– **setDrawRotation:**(NXCoord)*angle*

Disables rotation of the ClipView's coordinate system. The proper way to rotate objects in the document view is to perform the rotation in your document view's **drawSelf::** method. Returns **self**.

See also: – **rotate:**

setDrawSize::

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

Overrides the View method so that rescaling of the ClipView's coordinate system is reflected in the displayed document view. This method may redisplay the document view, and a **reflectScroll:** message may be sent to the ClipView's superview. Returns **self**.

See also: – **setDrawSize::** (View)

sizeTo::

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

Overrides the View method so that resizing of the ClipView's frame rectangle is reflected in the displayed document view. This method may redisplay the document view, and a **reflectScroll:** message may be sent to the ClipView's superview. Returns **self**.

See also: – **sizeTo::** (View)

translate::

– **translate:**(NXCoord)*x* **:**(NXCoord)*y*

Overrides the View method so that translation of the ClipView's coordinate system is reflected in the displayed document view. This method may redisplay the document view, and a **reflectScroll:** message may be sent to the ClipView's superview. Returns **self**.

See also: – **translate::** (View)

write:

– **write:**(NXTypedStream *)*stream*

Writes the ClipView and its document view to the typed stream *stream*. Returns **self**.

See also: – **read:**

Methods Implemented by a ClipView's Superview

reflectScroll:

– **reflectScroll:***aClipView*

Notifies the ClipView's superview that either the ClipView's bounds rectangle or the document view's frame rectangle has changed, and that any indicators of the scroll position need to be adjusted. ScrollView implements this method to update its Scroller.

scrollClip:to:

– **scrollClip:***aClipView* **to:**(const NXPoint *)*newOrigin*

Notifies the ClipView's superview that the ClipView needs to set its bounds rectangle origin to *newOrigin*. The ClipView's superview should then send the ClipView the **rawScroll:** message. This mechanism is provided so that the ClipView's superview can coordinate scrolling of multiple tiled ClipViews. Note that the default delegate is the ClipView's ScrollView, and doesn't respond to this method.

See also: – **rawScroll:** (ClipView)

Control

Inherits From: View : Responder : Object

Declared In: appkit/Control.h

Class Description

Control is an abstract superclass that provides three fundamental features for implementing user interface devices. First, as a subclass of View, Control allows the on-screen representation of the device to be drawn. Second, it receives and responds to user-generated events within its bounds by overriding Responder's **mouseDown:** method and providing a position in the responder chain. Third, it implements the **sendAction:to:** method to send an action message to the Control's target object. Subclasses of Control defined in the Application Kit are Button, Form, Matrix, NXBrowser, NXColorWell, Slider, Scroller, and TextField.

Target and Action

Target objects and action methods provide the mechanism by which Controls interact with other objects in an application. A target is an object that a Control has effect over. The target class defines an action method to enable its instances to respond to user input. An action method takes only one argument: the **id** of the sender. The sender may be either Control that sends the action message or another object that the target should treat as the sender. When it receives an action message, a target can return messages to the sender requesting additional information about its status. Control's **sendAction:to:** asks the Application object, NXApp, to send an action message to the Control's target object. The method used for this is Application's **sendAction:to:from:**. You can also set the target to **nil** and allow it to be determined at run time. When the target is **nil**, the Application object must look for an appropriate receiver. It conducts its search in a prescribed order, by following the responder chain until it finds an object that can respond to the message:

- It begins with the first responder in the key window and follows **nextResponder** links up the responder chain to the Window object. After the Window object, it tries the Window's delegate.
- If the main window is different from the key window, it then starts over with the first responder in the main window and works its way up the main window's responder chain to the Window object and its delegate.

- Next, it tries to respond itself. If the Application object can't respond, it tries its own delegate. NXApp and its delegate are the receivers of last resort.

Control provides methods for setting and using the target object and the action method. However, these methods require that a Control have an associated subclass of Cell that provides a target and an action, such as ActionCell and its subclasses.

Target objects and action methods demonstrate the close relationship between Controls and Cells. In most cases, a user interface device consists of an instance of a Control subclass paired with one or more instances of a Cell subclass. Each implements specific details of the user interface mechanism. For example, Control's **mouseDown:** method sends a **trackMouse:inRect:ofView:** message to a Cell, which handles subsequent mouse and keyboard events; a Cell sends a Control a **sendAction:to:** message in response to particular events. Control's **drawSelf::** method is implemented by sending a **drawSelf:inView:** message to the Cell. As another example, Control provides methods for setting and formatting its contents; these methods send corresponding messages to Cell, which actually owns the contents.

See the ActionCell class specification for more on the implementation of target and action behavior.

Creating New Controls

Since Control uses the Cell class to implement most of its actual functionality, you can usually implement a unique user interface device by creating a subclass of Cell or ActionCell rather than Control. A Control subclass doesn't have to use a Cell subclass to implement itself; Scroller and NXColorWell don't. However, such subclasses have to take care of details that Cell would otherwise handle. Specifically, they have to override methods designed to work with a Cell. What's more, they cannot be used in a Matrix—a subclass of Control designed specifically for managing multi-cell arrays such as radio buttons.

The **initFrame:** method is the designated initializer for the Control class. Override this method if you create a subclass of Control that performs its own initialization.

If your new Control uses a custom subclass of Cell, you'll probably also want to override Control's **setCellClass:** class method. Since Objective C does not support class variables, if you create a subclass of, for example, Button, and send **setCellClass:** to your subclass object to use a custom Cell, then all Buttons created after that will also use that Cell class. There are two ways to circumvent this problem. One is to reset the Cell class each time you create an instance of your Control subclass. The other is to override **setCellClass:** to store its own Cell class in a global variable and to use that in its **initFrame:** method as follows (note that in the **initialize** method MyCellSubClass checks itself to prevent its subclasses from inheriting a method that initializes them incorrectly):

```
static id myStoredCellClass;

+ initialize
{
    /* Class initialization code. */
    if (self == [MyCellSubclass class]) {
        myStoredCellClass = [MyCellSubclass class];  // Default class
    }
    return self;
}

+ setCellClass:classId
{
    myStoredCellClass = classId;
    return self;
}

- initFrame:(NXRect *)frameRect
{
    id oldCell;

    [super initFrame:frameRect];
    oldCell = [self setCell:[[myStoredCellClass alloc] init]];
    [oldCell free];
    /* other initialization code */

    return self;
}
```

Instance Variables

int **tag**;
id **cell**;
struct _conFlags {
    unsigned int **enabled**:1;
    unsigned int **editingValid**:1;
    unsigned int **ignoreMultiClick**:1;
    unsigned int **calcSize**:1;
} **conFlags**;

tag	Identifies the Control; used by View's **findViewWithTag:** method.
cell	The Control's Cell (if it has only one).
conFlags.enabled	YES if the Control is enabled; relevant for multi-cell controls only.
conFlags.editingValid	YES if editing has been validated.
conFlags.ignoreMultiClick	YES if the Control ignores double- or triple-clicks.
conFlags.calcSize	YES if the cell should recalculate its size and location before drawing.

Method Types

Initializing and freeing a Control	– initFrame: – free
Setting the Control's Cell	+ setCellClass: – setCell: – cell
Enabling and disabling the Control	– isEnabled – setEnabled:
Identifying the selected Cell	– selectedCell – selectedTag
Setting the Control's value	– setFloatValue: – floatValue – setDoubleValue: – doubleValue – setIntValue: – intValue – setStringValue: – setStringValueNoCopy: – setStringValueNoCopy:shouldFree: – stringValue
Interacting with other Controls	– takeDoubleValueFrom: – takeFloatValueFrom: – takeIntValueFrom: – takeStringValueFrom:

Formatting text	– setAlignment: – alignment – setFont: – font – setFloatingPointFormat:left:right:
Managing the field editor	– abortEditing – currentEditor – validateEditing
Managing the cursor	– resetCursorRects
Resizing the Control	– calcSize – sizeTo:: – sizeToFit
Displaying the Control and Cell	– drawCell: – drawCellInside: – drawSelf:: – selectCell: – update – updateCell: – updateCellInside:
Target and action	– setAction: – action – setTarget: – target – setContinuous: – isContinuous – sendAction:to: – sendActionOn:
Assigning a tag	– setTag: – tag
Tracking the mouse	– ignoreMultiClick: – mouseDown: – mouseDownFlags
Archiving	– read: – write:

Class Methods

setCellClass:

+ **setCellClass:***classId*

This abstract method does nothing. It's implemented by subclasses of Control, which use this method to set the class of their Cells. Returns **self.**

Instance Methods

abortEditing

– **abortEditing**

Terminates and discards any editing of text displayed by the receiving Control. Returns **self**, or **nil** if no editing was going on in the receiving Control. This method doesn't redisplay the old value of the Control.

See also: – **endEditingFor:** (Window), – **validateEditing**

action

– (SEL)**action**

Returns the action message sent by the Control's Cell, or the default action message for a Control with multiple Cells (such as a Matrix or Form). To retrieve the action message, this method sends an **action** message to the Cell. For Controls with multiple Cells, it's better to get the action message for a particular Cell using:

```
someAction = [[theControl selectedCell] action];
```

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

alignment

– (int)**alignment**

Returns the alignment mode of the text in the Control's Cell. The return value can be one of three constants: NX_LEFTALIGNED, NX_CENTERED or NX_RIGHTALIGNED.

See also: – **setAlignment:**

calcSize

– **calcSize**

Recomputes any internal sizing information for the Control, if necessary, by invoking its Cell's **calcDrawInfo:** method. This method doesn't actually draw. It can be used for more sophisticated sizing operations as well (for example, Form). **calcSize** is automatically invoked whenever the Control is displayed and something has changed; you need never invoke it. Returns **self**.

See also: – **calcSize** (Matrix, Form), – **sizeToFit**

cell

– **cell**

Returns the Control's Cell. You should use **selectedCell** in the action method of the target of the Control, since a Control may have multiple Cells.

See also: – **selectedCell**

currentEditor

– **currentEditor**

If the receiving Control is being edited (that is, has a Text object acting as its editor, and is the first responder in its Window), this method returns the Text object being used to perform that editing. If the Control isn't being edited, this method returns **nil**.

See also: – **abortEditing**, – **validateEditing**

doubleValue

– (double)**doubleValue**

Returns the value of the Control's selected Cell as a double-precision floating point number. If the Control contains many cells (for example, Matrix), then the value of the currently **selectedCell** is returned. If the Control is in the process of editing the affected Cell, then **validateEditing** is invoked before the value is extracted and returned.

See also: – **setDoubleValue:**, – **floatValue**, – **intValue**, – **stringValue**

drawCell:

– **drawCell:***aCell*

If *aCell* is the cell used to implement this Control, then the Control is displayed. This method is provided primarily to support a consistent set of methods between Controls with single and multiple Cells, since a Control with multiple Cells needs to be able to draw a single Cell at a time. Returns **self**.

See also: – **updateCell:**, – **drawCellInside:**, – **updateCellInside:**, – **drawCell:** (Matrix)

drawCellInside:

– **drawCellInside:***aCell*

Draws the inside of a Control (the area within a bezel or border). This method invokes Cell's **drawInside:inView:** method. **drawCellInside:** is used by **setStringValue:** and similar content-setting methods to provide a minimal update of the Control when its value is changed. Returns **self**.

See also: – **drawCell:**, – **drawInside:inView:** (Cell), – **drawCellInside:** (Matrix), – **updateCellInside:**

drawSelf::

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

Draws the Control. This method invokes the **drawSelf:inView:** method of the Control's Cell. You must override this method if you have a Control with multiple Cells. Returns **self**.

See also: – **drawSelf:inView:** (Cell)

floatValue

– (float)**floatValue**

Returns the value of the Control's selected Cell as a single-precision floating point number. See **doubleValue** for more details.

See also: – **setFloatValue:**, – **doubleValue**, – **intValue**, – **stringValue**

font

– **font**

Returns the Font object used to draw the text (if any) of the Control's Cell.

See also: – **setFont:**

free

– **free**

Frees the memory used by the Control and its Cells. Aborts editing if the text of the Control was currently being edited. Returns **nil**.

See also: – **free** (View)

ignoreMultiClick:

– **ignoreMultiClick:**(BOOL)*flag*

Sets the Control to ignore multiple clicks if *flag* is YES. By default, double-clicks (and higher order clicks) are treated the same as single clicks. You can use this method to "debounce" a Control, so that it won't inadvertently send its action message twice when double-clicked. Returns **self**.

initFrame:

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

Initializes and returns the receiver, a new instance of Control, by setting the value pointed to by *frameRect* as its frame rectangle. Makes the new instance an opaque View. Since Control is an abstract class, messages to perform this method should appear only in subclass methods; that is, there should always be a more specific designated initializer for the subclass. **initFrame:** is the designated initializer for the Control class.

intValue

– (int)**intValue**

Returns the value of the Control's selected Cell as an integer (see **doubleValue** for more details).

See also: – **setIntValue:**, – **doubleValue**, – **floatValue**, – **stringValue**

isContinuous

– (BOOL)**isContinuous**

Returns YES if the Control's Cell continuously sends its action message to its target during mouse tracking.

See also: – **setContinuous:**

isEnabled

– (BOOL)**isEnabled**

Returns YES if the Control is enabled, NO otherwise.

See also: – **setEnabled:**

mouseDown:

– **mouseDown:**(NXEvent *)*theEvent*

Highlights the Control, and sends **trackMouse:inRect:ofView:** to the Control's Cell (or whichever Cell the mouse event occured in if the Control has multiple Cells). This method is invoked when the mouse button goes down while the cursor is within the bounds of the Control. The Control's Cell tracks the cursor until it goes outside the bounds, at which time the Control is unhighlighted. If the cursor goes back into the bounds, then the Control highlights again and its Cell starts tracking again. This behavior continues until the mouse button goes up. If it goes up with the cursor in the Control, the state of the Control is changed, and the action message is sent to the target with **sendAction:to:**. If the mouse button goes up with the cursor outside the Control, no action message is sent. Returns **self**.

See also: – **trackMouse:inRect:ofView:** (Cell), – **sendAction:to:**

mouseDownFlags

– (int)**mouseDownFlags**

Returns the event flags (for example, NX_SHIFTMASK) that were in effect at the beginning of mouse tracking. The flags are valid only in the action method invoked upon the Control's target.

See also: – **mouseDownFlags** (Cell), – **sendAction:to:**

read:

– **read:**(NXTypedStream *)*stream*

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

resetCursorRects

– **resetCursorRects**

Reestablishes the cursor rectangles for the Control's Cell (or Cells). If the Cell displays text, and the text in the Cell is selectable, then **resetCursorRect:inView:** is sent to the Cell. **resetCursorRect:inView:** in turn, sends **addCursorRect:cursor:** back to the Control, so that the cursor will change to an I-beam when it enters the Cell's rectangle. Returns **self**.

See also: – **resetCursorRect:inView:** (Cell), – **addCursorRect:cursor:** (View)

selectCell:

– **selectCell:***aCell*

If *aCell* is a Cell of the receiving Control and is unselected, this method selects *aCell* and redraws the Control. Returns **self**.

selectedCell

– **selectedCell**

Returns the Control's selected Cell. The target of the Control should use this method when it wants to get the Cell of the sending Control. Note that even though the **cell** method will return the same value for Controls with only a single Cell, it's strongly suggested that this method be used since it will work for Controls with either a single or multiple Cells.

See also: – **sendAction:to:**, – **selectedCell** (Matrix)

selectedTag

– (int)**selectedTag**

Returns the tag of the Control's selected Cell. This is equivalent to:

```
myTag = [[theControl selectedCell] tag];
```

Returns –1 if there is no selected Cell. The Cell's tag can be set with ActionCell's **setTag:** method. You should only use the **setTag:** and **tag** methods in conjunction with **findViewWithTag:**. When you set the tag of a Control with a single Cell in Interface Builder, it sets both the tags of both Control and Cell as a convenience.

See also: – **sendAction:to:**

sendAction:to:

– **sendAction:**(SEL)*theAction* **to:***theTarget*

Sends a **sendAction:to:from:** message to NXApp, which in turn sends a message to *theTarget* to perform *theAction*. **sendAction:to:from:** adds the Control as *theAction*'s only argument. If *theAction* is NULL, no message is sent. **sendAction:to:** is invoked primarily by Cell's **trackMouse:inRect:ofView:**

If *theTarget* is **nil**, NXApp looks for an object that can respond to the message by following the responder chain, as detailed in the class description.

Returns **nil** if no object that responds to *theAction* could be found; otherwise returns **self**.

See also: – **action**, – **target**, – **trackMouse:inRect:ofView:** (Cell), – **sendAction:to:from:** (Application)

sendActionOn:

– (int)**sendActionOn:**(int)*mask*

Uses *mask* to record the events that cause **sendAction:to:** to be invoked during tracking of the mouse, which is performed in Cell's **trackMouse:inRect:ofView:**. Returns the old event mask.

See also: – **sendAction:to:**, – **sendActionOn:** (Cell), – **trackMouse:inRect:ofView:** (Cell)

setAction:

– **setAction:**(SEL)*aSelector*

Makes *aSelector* the Control's action method. If *aSelector* is NULL, then no action messages will be sent from the Control. Returns **self**.

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

setAlignment:

– **setAlignment:**(int)*mode*

Sets the alignment mode of the text in the Control's Cell, or of all the Control's Cells if it has more than one, and redraws the Control. *mode* should be one of: NX_LEFTALIGNED, NX_CENTERED or NX_RIGHTALIGNED. Returns **self**.

See also: – **alignment**

setCell:

– **setCell:***aCell*

Sets the Cell of the Control to be *cell*. Use this method with great care as it can irrevocably damage your Control; specifically, you should only use this method in initializers for subclasses of Control. Returns the old Cell.

setContinuous:

– **setContinuous:**(BOOL)*flag*

Sets whether the Control will continuously send its action message to its target as the mouse is tracked. Returns **self**.

See also: – **setContinuous:** (ButtonCell, SliderCell), – **sendActionOn:**

setDoubleValue:

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

Sets the value of the Control's selected Cell to be *aDouble* (a double-precision floating point number). If the affected Cell is being edited, that editing is aborted and the value being typed is discarded in favor of *aDouble*. If autodisplay is on, then the Cell's inside (the area within a bezel or border) is redrawn. Returns **self**.

See also: – **doubleValue**, – **setFloatValue:**, – **setIntValue:**, – **setStringValue:**, – **abortEditing**, – **drawInside:inView:** (Cell), – **isAutodisplay** (View), – **setAutodisplay:** (View)

setEnabled:

– **setEnabled:**(BOOL)*flag*

Sets whether the Control is active or not (that is, whether it tracks the mouse and sends its action to its target). If *flag* is NO, any editing is aborted. Redraws the entire Control if autodisplay is on. Subclasses may want to override this to redraw only a portion of the Control when the enabled state changes (Button and Slider do this). Returns **self**.

See also: – **setEnabled:** (Cell), – **isAutodisplay** (View), – **setAutodisplay:** (View)

setFloatValue:

– **setFloatValue:**(float)*aFloat*

Same as **setDoubleValue:**, but sets the value as a single-precision floating point number. Returns **self**.

See also: – **floatValue**, – **setDoubleValue:**, – **setIntValue:**, – **setStringValue:**

setFloatingPointFormat:left:right:

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

Sets the autoranging and floating point number format of the Control's Cell, so that at most *leftDigits* are displayed to the left of the decimal point, and *rightDigits* to the right. If the Control has more than one Cell, they're all affected. See the description of this method in the Cell class specification for more detail. This method doesn't redraw the Control. **setFloatingPointFormat:left:right:** affects only subsequent invocations of **setFloatValue:**. Returns **self**.

See also: – **setFloatingPointFormat:left:right:** (Cell)

setFont:

– **setFont:***fontObject*

Sets the Font object used to draw the text (if any) in the Control's Cell, or in all the Cells if the Control has more than one. You only need to use this method if you don't want to use the user's default system font (as set by the user in the Preferences application). If autodisplay is on, then the inside of the Cell is redrawn. Returns **self**.

See also: – **font**, – **isAutodisplay** (View), – **setAutodisplay:** (View)

setIntValue:

– **setIntValue:**(int)*anInt*

Same as **setDoubleValue:**, but sets the value as an integer. Returns **self**.

See also: – **intValue**, – **setDoubleValue:**, – **setFloatValue:**, – **setStringValue:**

setStringValue:

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

Same as **setDoubleValue:**, but sets the value as a string by copying it from *aString*. Returns **self**.

See also: – **stringValue**, – **setStringValueNoCopy:**,
– **setStringValueNoCopy:shouldFree:**, – **setDoubleValue:**, – **setFloatValue:**,
– **setIntValue:**

setStringValueNoCopy:

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

Like **setStringValue:**, but doesn't copy the string. Returns **self**.

See also: – **stringValue**, – **setStringValue:**, – **setStringValueNoCopy:**,
– **setStringValueNoCopy:shouldFree:**, – **setDoubleValue:**, – **setFloatValue:**,
– **setIntValue:**

setStringValueNoCopy:shouldFree:

– **setStringValueNoCopy:**(char *)*aString* **shouldFree:**(BOOL)*flag*

Like **setStringValueNoCopy:**, but lets you specify whether the string should be freed when the Control is freed. Returns **self**.

See also: – **stringValue**, – **setStringValue:** – **setStringValueNoCopy:**,
– **setDoubleValue:**, – **setFloatValue:**, – **setIntValue:**

setTag:

– **setTag:**(int)*anInt*

Makes *anInt* the receiving Control's tag. Doesn't affect the Control's Cell. Returns **self**.

See also: – **tag**, – **selectedTag**, – **findViewWithTag:** (View), – **setTag:** (Cell)

setTarget:

– **setTarget:***anObject*

Sets the target for the action message of the Control's Cell. Returns **self**.

If *anObject* is **nil**, then when an action message is sent, NXApp looks for an object that can respond to the message by following the responder chain, as detailed in the class description.

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

sizeTo::

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

Changes the width and the height of the Control's frame. Redisplays the Control if autodisplay is on. Returns **self**.

See also: – **isAutodisplay** (View), – **setAutodisplay:** (View)

sizeToFit

– **sizeToFit**

Changes the width and the height of the Control's frame so that they are the minimum needed to contain the Cell. If the Control has more than one Cell, then you must override this method. Returns **self**.

See also: – **sizeToFit** (Matrix), – **sizeToCells** (Matrix)

stringValue

– (const char *)**stringValue**

Returns the value of the Control's selected Cell as a string. If the Control is in the process of editing the affected Cell, then **validateEditing** is invoked before the value is extracted and returned.

See also: – **setStringValue:**, – **doubleValue**, – **floatValue**, – **intValue**

tag

– (int)**tag**

Returns the receiving Control's tag (not the tag of the Control's Cell).

See also: – **setTag:**, – **selectedTag**, – **tag** (Cell)

takeDoubleValueFrom:

– **takeDoubleValueFrom:***sender*

Sets the double-precision floating-point value of the receiving Control's selected Cell to the value obtained by sending a **doubleValue** message to *sender*. Returns **self**.

This method can be used in action messages between Controls. It permits one Control (the sender) to affect the value of another Control (the receiver) by invoking this method in an action message to the receiver. For example, a TextField can be made the target of a Slider. Whenever the Slider is moved, it will send a **takeDoubleValueFrom:** message to the TextField. The TextField will then get the Slider's floating-point value, turn it into a text string, and display it, thus tracking the value of the Slider.

See also: – **setDoubleValue:**, – **doubleValue**

takeFloatValueFrom:

– **takeFloatValueFrom:***sender*

Sets the single-precision floating-point value of the receiving Control's selected Cell to the value obtained by sending a **floatValue** message to *sender*. Returns **self**.

See **takeDoubleValueFrom:** for an example.

See also: – **setFloatValue:**, – **floatValue**

takeIntValueFrom:

– **takeIntValueFrom:***sender*

Sets the integer value of the receiving Control's selected Cell to the value returned by sending an **intValue** message to *sender*. Returns **self**.

See **takeDoubleValueFrom:** for an example.

See also: – **setIntValue:**, – **intValue**

takeStringValueFrom:

– **takeStringValueFrom:***sender*

Sets the character string of the receiving Control's selected Cell to a string obtained by sending a **stringValue** message to *sender*. Since this is an action method, there is no alternate like **takeStringValueFrom:noCopy:**. Returns **self**.

See **takeDoubleValueFrom:** for an example.

See also: – **stringValue**, – **setStringValue:**

target

– **target**

Returns the target for the action message of the Control's cell, or the Control's target for a Control with multiple Cells. If **nil**, then any action messages sent by the Control will be sent up the responder chain, as detailed in the Class Description.

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

update

– **update**

If autodisplay is enabled, sends a **display** message to itself. Otherwise it simply sets a flag indicating that the Control needs to be displayed. This method also makes sure that **calcSize** is performed. Returns **self**.

See also: – **updateCell:**, – **updateCellInside:**

updateCell:

– **updateCell:***aCell*

If *aCell* is a Cell used to implement this Control, and if autodisplay is on, then draws the Control's Cell; otherwise, sets the **needsDisplay** and **calcSize** flags to YES. Returns **self**.

See also: – **update**, – **updateCellInside:**, – **isAutodisplay** (View), – **setAutodisplay:** (View)

updateCellInside:

– **updateCellInside:***aCell*

If *aCell* is a Cell used to implement this Control, and if autodisplay is on, draws the inside portion of the Cell; otherwise sets the **needsDisplay** flag to YES. Returns **self**.

See also: – **update**, – **updateCell:**, – **isAutodisplay** (View), – **setAutodisplay:** (View)

validateEditing

– **validateEditing**

Causes the value of the Control's selected Cell to be set to the value of the field being edited, if any. "Being edited" does not necessarily mean that a user is typing; if a field (for example, a TextField object) has the application's global Text object acting in its place as first responder, then the field is considered as being edited. This method is invoked automatically from **stringValue**, **intValue**, and other similar methods, so that a partially edited field's actual value will be correctly returned by those methods. Returns **self**.

This method doesn't end editing; to do that, invoke Window's **endEditingFor:** or **abortEditing**.

See also: – **endEditingFor:** (Window), – **abortEditing**

write:

– **write:**(NXTypedStream *)*stream*

Writes the Control to the typed stream *stream*.

See also: – **read:**

Font

Inherits From: Object

Declared In: appkit/Font.h

Class Description

The Font class provides objects that correspond to PostScript fonts. Each Font object records a font's name, size, style, and matrix. When a Font object receives a **set** message, it establishes its font as the current font in the Window Server's current graphics state.

For a given application, only one Font object is created for a particular PostScript font. When the Font class object receives a message to create a new object for a particular font, it first checks whether one has already been created for that font. If so, it returns existing object; otherwise, it creates a new object and returns it. To implement this sharing of Font objects, the Font class provides special instantiation methods (the **new...** methods, **userFixedPitchFontOfSize:matrix:**, and so on); use these methods, not **alloc** or **allocFromZone:**.

This sharing Font objects minimizes the number of objects created. It also implies that no one object in your application can know whether it has the only reference to a particular Font object. Thus, Font objects shouldn't be freed; Font's **free** method simply returns **self**.

Instance Variables

```
char *name;
float size;
int style;
float *matrix;
int fontNum;
NXFaceInfo *faceInfo;
id otherFont;
struct _fFlags {
        unsigned int  isScreenFont:1;
} fFlags;
```

name	The font's name.
size	The font's size.
style	The font's style.
matrix	The font's matrix.
fontNum	The user object referring to this font.
faceInfo	The font's face information.
otherFont	The associated screen font for this font.
fFlags.isScreenFont	True if the font is a screen font.

Method Types

Initializing the Class object	+ initialize + useFont:
Creating and freeing a Font object	+ newFont:size: + newFont:size:matrix: + newFont:size:style:matrix: + boldSystemFontOfSize:matrix: + userFixedPitchFontOfSize:matrix: + userFontOfSize:matrix: + systemFontOfSize:matrix: – free
Querying the Font object	– displayName – familyName – name – fontNum – getWidthOf: – hasMatrix – matrix – metrics – pointSize – readMetrics: – screenFont – style

Setting the font	– set – setStyle: + setUserFixedPitchFont: + setUserFont:
Archiving	– awake – finishUnarchiving – read: – write:

Class Methods

allocFromZone:

+ **allocFromZone:**(NXZone *)*zone*

Creates an uninitialized Font object in the specified zone. Don't use this method to create a Font; instead, use **newFont:size:** or one of the other Font creation methods listed in "Method Types" above.

See also: + **newFont:size:style:matrix:**, + **newFont:size:matrix:**, + **newFont:size:**

boldSystemFontOfSize:matrix:

+ **boldSystemFontOfSize:**(float)*fontSize* **matrix:**(const float *)*fontMatrix*

Returns the Font object representing the bold system font of size *fontSize* and matrix *fontMatrix*. The bold system font is used for text in attention panels, window titles, and so on. If *fontSize* is 0, the size as recorded in the Preferences application's General Preferences display is used. *fontMatrix* can be NX_IDENTITYMATRIX or NX_FLIPPEDMATRIX. (See **newFont:size:style:matrix:** for more information on font matrices.)

This method raises the NX_unavailableFont exception if a suitable Font object can't be found.

See also: + **systemFontOfSize:matrix:**, + **userFixedPitchFontOfSize:matrix:**, + **userFontOfSize:matrix:**

initialize

+ **initialize**

Initializes the Font class object. The class object receives an **initialize** message before it receives any other message. You never send an **initialize** message directly.

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

newFont:size:

+ **newFont:**(const char *)*fontName* **size:**(float)*fontSize*

Returns a Font object for font *fontName* of size *fontSize*. This method invokes the **newFont:size:style:matrix:** method with the style set to 0 and the matrix set to NX_FLIPPEDMATRIX.

See also: + **newFont:size:style:matrix:**, + **newFont:size:matrix:**

newFont:size:matrix:

+ **newFont:**(const char *)*fontName*
  size:(float)*fontSize*
  matrix:(const float *)*fontMatrix*

Returns a Font object for font *fontName* of size *fontSize*. This method invokes the **newFont:size:style:matrix:** method with the style set to 0.

See also: + **newFont:size:style:matrix:**, + **newFont:size:**

newFont:size:style:matrix:

+ **newFont:**(const char *)*fontName*
  size:(float)*fontSize*
  style:(int)*fontStyle*
  matrix:(const float *)*fontMatrix*

Returns a Font object for font *fontName*, of size *fontSize*, and matrix *fontMatrix*. *fontStyle* is currently ignored. If an appropriate Font object was previously created, it's returned; otherwise, a new one is created and returned. If an error occurs, this method returns **nil**. This is the designated **new...** method for the Font class.

There are two constants available for the *fontMatrix* parameter:

- NX_IDENTITYMATRIX. Use the identity matrix.

- NX_FLIPPEDMATRIX. Use a flipped matrix. (Appropriate for a flipped View like the Text object.)

The *fontStyle* parameter is stored in the Font object, and is preserved by the FontManager's **convertFont:** method, but is not used by the Application Kit. It can be used to store application-specific font information.

Note: If this method is invoked from a subclass (through a message to **super**), a new object is always created. Thus, your subclass should institute its own system for sharing Font objects.

See also: + **newFont:size:matrix:**, + **newFont:size:**

setUserFixedPitchFont:

+ **setUserFixedPitchFont:**(Font *)*aFont*

Sets the fixed-pitch font that's used by default in the application. This method is intended for an application that wants to override the default fixed-pitch font as recorded in the Preferences application's General Preferences display.

See also: + **userFixedPitchFontOfSize:matrix:**, + **setUserFont:**

setUserFont:

+ **setUserFont:**(Font *)*aFont*

Sets the standard font that's used by default in the application. This method is intended for an application that wants to override the default standard font as recorded in the Preferences application's General Preferences display.

See also: + **userFontOfSize:matrix:**, + **setUserFixedPitchFont:**

systemFontOfSize:matrix:

+ **systemFontOfSize:**(float)*fontSize* **matrix:**(const float *)*fontMatrix*

Returns the Font object representing the system font of size *fontSize* and matrix *fontMatrix*. The system font is used for text in attention panels, menus, and so on. If *fontSize* is 0, the size as recorded in the Preferences application's General Preferences display is used. *fontMatrix* can be NX_IDENTITYMATRIX or NX_FLIPPEDMATRIX. (See **newFont:size:style:matrix:** for more information on font matrices.)

This method raises the NX_unavailableFont exception if a suitable Font object can't be found.

See also: + **boldSystemFontOfSize:matrix:**, + **userFixedPitchFontOfSize:matrix:**, + **userFontOfSize:matrix:**

useFont:

+ **useFont:**(const char *)*fontName*

Registers that the font identified by *fontName* is used in the document. Returns **self**.

The Font class object keeps track of the fonts that are being used in a document. It does this by registering the font whenever a Font object receives a **set** message. When a document is called upon to generate a conforming PostScript language version of its text (such as during printing), the Font class provides the list of fonts required for the **%%DocumentFonts** comment. (See *Document Structuring Conventions* by Adobe Systems Inc.)

The **useFont:** method augments this system by providing a way to register fonts that are included in the document but not set using Font's **set** method. For example, you might set a font by executing the **setfont** operator within a function created by **pswrap**. In such a case, make sure to pair the use of the font with a **useFont:** message to register the font with the Font class object.

See also: – **set**

userFixedPitchFontOfSize:matrix:

+ **userFixedPitchFontOfSize:**(float)*fontSize* **matrix:**(const float *)*fontMatrix*

Returns the Font object representing the application's fixed-pitch font of size *fontSize* and matrix *fontMatrix*. If *fontSize* is 0, the size as recorded in the Preferences application's General Preferences display is used. *fontMatrix* can be NX_IDENTITYMATRIX or NX_FLIPPEDMATRIX. (See **newFont:size:style:matrix:** for more information on font matrices.)

This method raises the NX_unavailableFont exception if a suitable Font object can't be found.

See also: + **setUserFixedPitchFont:**, + **boldSystemFontOfSize:matrix:**, + **systemFontOfSize:matrix:**, + **userFontOfSize:matrix:**

userFontOfSize:matrix:

+ **userFontOfSize:**(float)*fontSize* **matrix:**(const float *)*fontMatrix*

Returns the Font object representing the application's standard font of size *fontSize* and matrix *fontMatrix*. If *fontSize* is 0, the size as recorded in the Preferences application's General Preferences display is used. *fontMatrix* can be NX_IDENTITYMATRIX or NX_FLIPPEDMATRIX. (See **newFont:size:style:matrix:** for more information on font matrices.)

This method provides an easy way to determine the user's font preference, which you can use to initialize new documents.

userFontOfSize:matrix: raises the NX_unavailableFont exception if a suitable Font object can't be found.

See also: + **setUserFixedPitchFont:**, + **boldSystemFontOfSize:matrix:**, + **systemFontOfSize:matrix:**, + **userFixedPitchFontOfSize:matrix:**

Instance Methods

awake

– **awake**

Reinitializes the Font object after it's been read in from a stream.

An **awake** message is automatically sent to each object of an application after all objects of that application have been read in. You never send **awake** messages directly. The **awake** message gives the object a chance to complete any initialization that **read:** couldn't do. If you override this method in a subclass, the subclass should send this message to its superclass:

```
[super awake];
```

Returns **self**.

See also: – **read:**, – **write:**, – **finishUnarchiving**

displayName

– (const char *)**displayName**

Returns the full name of the font. For example, the font named "Futura-CondExtraBoldObl" returns the display name "Futura Condensed Extra Bold Oblique".

See also: – **familyName**, – **name**

familyName

– (const char *)**familyName**

Returns the name of the font's family. For example, the font named "Futura-CondExtraBoldObl" returns the family name "Futura".

See also: – **displayName**, – **name**

finishUnarchiving

– **finishUnarchiving**

A **finishUnarchiving** message is sent after the Font object has been read in from a stream. This method checks if a Font object for the particular PostScript font already exists. If so, **self** is freed and the existing object is returned.

See also: – **read:**, – **write:**, – **awake**

fontNum

– (int)**fontNum**

Returns the PostScript user object that corresponds to this font. The Font object must set the font in the Window Server before this method will return a valid user object. Sending a Font object the **set** message sets the font in the Window Server. The **fontNum** method returns 0 if the Font object hasn't previously received a **set** message or if the font couldn't be set.

See also: – **set**, **DPSDefineUserObject**()

free

– **free**

Has no effect. Since only one Font object is allocated for a particular font, and since you can't be sure that you have the only reference to a particular Font object, a Font object shouldn't be freed.

getWidthOf:

– (float)**getWidthOf:**(const char *)*string*

Returns the width of *string* using this font. This method has better performance than the Window Server routine **PSstringwidth**().

hasMatrix

– (BOOL)**hasMatrix**

Returns YES if the Font object's matrix is different from the identity matrix, NX_IDENTITYMATRIX; otherwise, returns NO.

See also: + **newFont:size:style:matrix:**, – **matrix**

matrix

– (const float *)**matrix**

Returns a pointer to the matrix for this font.

See also: – **hasMatrix**

metrics

– (NXFontMetrics *)**metrics**

Returns a pointer to the NXFontMetrics record for the font. See the header file **appkit/afm.h** for the structure of an NXFontMetrics record.

See also: – **readMetrics:**

name

– (const char *)**name**

Returns the font's name, as would be used in a PostScript language program.

See also: – **displayName**, – **familyName**

pointSize

– (float)**pointSize**

Returns the size of the font in points.

read:

– **read:**(NXTypedStream *)*stream*

Reads the Font object's instance variables from *stream*. A **read:** message is sent in response to archiving; you never send this message.

See also: – **write:**, – **read:** (Object)

readMetrics:

– (NXFontMetrics *)**readMetrics:**(int)*flags*

Returns a pointer to the NXFontMetrics record for this font. The *flags* argument determines which fields of the record will be filled in. *flags* is built by ORing together constants such as NX_FONTHEADER, NX_FONTMETRICS, and NX_FONTWIDTHS. See the header file **appkit/afm.h** for the complete list of constants and for the structure of the NXFontMetrics record.

See also: – **metrics**

screenFont

– **screenFont**

Provides the screen font corresponding to this font. If the receiver represents a printer font, this method returns the Font object for the associated screen font (or **nil** if one doesn't exist). If the receiver represents a screen font, it simply returns **self**.

set

– **set**

Makes this font the current font in the current graphics state. Returns **self**.

When a Font object receives a **set** message, it registers with the Font class object that its PostScript font has been used. In this way, the Application Kit, when called upon to generate a conforming PostScript language document file, can list the fonts used within a document. (See *Document Structuring Conventions* by Adobe Systems Inc.) If the application uses fonts without sending set messages (say through including an EPS file), such fonts must be registered by sending the class object a **useFont:** message.

See also: + **useFont:**

setStyle:

– **setStyle:**(int)*aStyle*

Sets the Font's style. Setting a style isn't recommended but is minimally supported—a Font object's style isn't interpreted in any way by the Application Kit. You can use it for your own non-PostScript language font styles (a drop-shadow style, for example).

Be very careful using this method since it causes the Font to stop being shared. You must reassign the pointer to the Font to the return value of **setStyle:**.

```
font = [font setStyle:12];
```

Returns **self**.

See also: – **style**

style

– (int)**style**

Returns the style of the font. For Font objects created by the Application Kit, this method returns 0.

See also: – **setStyle:**

write:

– **write:**(NXTypedStream *)*stream*

Writes the Font object's instance variables to *stream*. A **write:** message is sent in response to archiving; you never send this message directly.

See also: – **read:**, – **write:** (Object)

FontManager

Inherits From: Object

Declared In: appkit/FontManager.h

Class Description

The FontManager is the center of activity for font conversion. It accepts messages from font conversion user-interface objects (such as the Font menu or the Font panel) and appropriately converts the current font in the selection by sending a **changeFont:** message up the responder chain. When an object receives a **changeFont:** message, it should query the FontManager (by sending it a **convertFont:** message), asking it to convert the font in whatever way the user has specified. Thus, any object containing a font that can be changed should respond to the **changeFont:** message by sending a **convertFont:** message back to the FontManager for each font in the selection.

To use the FontManager, you simply insert a Font menu into your application's menu. This is most easily done with Interface Builder, but, alternatively, you can send a **getFontMenu:** message to the FontManager and then insert the menu that it returns into the application's main menu. Once the Font menu is installed, your application automatically gains the functionality of both the Font menu and the Font panel.

The FontManager's delegate can restrict which font names will be appear in the FontPanel. See "Methods Implemented by the Delegate" near the end of this class specification for more information.

The FontManager can be used to convert a font or find out the attributes of a font. It can also be overridden to convert fonts in some application-specific manner. The default implementation of font conversion is very conservative: The font isn't converted unless all traits of the font can be maintained across the conversion.

Instance Variables

```
id panel;
id menu;
SEL action;
int whatToDo;
NXFontTraitMask traitToChange;
id selFont;
struct _fmFlags {
    unsigned int multipleFont:1;
    unsigned int disabled:1;
} fmFlags;
```

panel	The Font panel.
menu	The Font menu.
action	The action to send.
whatToDo	What to do when a **convertFont:** message is received.
traitToChange	The trait to change if **whatToDo** == NX_CHANGETRAIT.
selFont	The font of the current selection.
fmFlags.multipleFont	True if the current selection has multiple fonts.
fmFlags.disabled	True if the Font panel and menu are disabled.

Method Types

Creating the FontManager	+ new
Converting fonts	– convertFont: – convertWeight:of: – convert:toFace: – convert:toFamily: – convert:toSize: – convert:toHaveTrait: – convert:toNotHaveTrait: – findFont:traits:weight:size: – getFamily:traits:weight:size:ofFont:

Setting parameters	– setAction: + setFontPanelFactory: + setFontManagerFactory: – setSelFont:isMultiple: – setEnabled:
Querying parameters	– action – availableFonts – getFontMenu: – getFontPanel: – isMultiple – selFont – isEnabled
Target and action methods	– modifyFont: – addFontTrait: – removeFontTrait: – modifyFontViaPanel: – orderFrontFontPanel: – sendAction
Assigning a delegate	– setDelegate: – delegate
Archiving the FontManager	– finishUnarchiving

Class Methods

alloc

Disables the inherited **alloc** method to prevent multiple FontManagers from being created. There's only one FontManager object for each application; you access it using the **new** method. Returns an error message.

See also: + **new**

allocFromZone:

Disables the inherited **allocFromZone** method to prevent multiple FontManagers from being created. There's only one FontManager object for each application; you access it using the **new** method. Returns an error message.

See also: + **new**

new

+ **new**

Returns a FontManager object. An application has no more than one FontManager object, so this method either returns the previously created object (if it exists) or creates a new one. This is the designated **new** method for the FontManager class.

setFontManagerFactory:

+ **setFontManagerFactory:***classId*

Sets the class object that will be used to create the font manager; thus allowing you to specify a class of your own. When the FontManager class object receives a **new** message, it creates an instance of the specified class, if no instance already exists. If no class has been specified, the **new** method creates an instance of the FontManager class.

As a consequence of this implementation, your class shouldn't implement the **new** method. Instead, initialization code should be place in the **init** method.

The **setFontManagerFactory:** method must be invoked before your application's main nib file is loaded. Returns **self**.

See also: – **setFontPanelFactory:**

setFontPanelFactory:

+ **setFontPanelFactory:***classId*

Sets the class object that's used to create the FontPanel object when the user chooses the Font Panel command from the Font menu and no such panel has yet been created. Unless you use this method to specify another class, the FontPanel class will be used. Returns **self**.

See also: – **setFontManagerFactory:**

Instance Methods

action

– (SEL)**action**

Returns the action that's sent to the first responder when the user selects a new font from the Font panel or from the Font menu.

See also: – **setAction:**

addFontTrait:

– **addFontTrait:***sender*

Causes the FontManager's action message (by default, **changeFont:**) to be sent up the responder chain. When the responder replies with a **convertFont:** message, the font is converted to add the trait specified by *sender*.

Before the action message is sent up the responder chain, the FontManager sets its **traitToChange** variable to the value returned by sending *sender* a **selectedTag** message. The FontManager also sets its **whatToDo** variable to NX_ADDTRAIT. When the **convertFont:** message is received, the FontManager converts the supplied font by sending itself a **convert:toHaveTrait:** message.

See also: – **removeFontTrait:**, – **convertFont:**, – **convert:toHaveTrait:**, – **selectedTag** (Control)

availableFonts

– (char **)**availableFonts**

Returns by reference a NULL-terminated list of NULL-terminated PostScript font names of all the fonts available for use by the Window Server. The returned names are suitable for creating new Fonts using the **newFont:size:** class method of the Font class. The fonts are not in any guaranteed order, but no font name is repeated in the list. It's the sender's responsibility to free the list when finished with it.

See also: + **newFont:size:** (Font)

convert:toFace:

– **convert:***fontObj* **toFace:**(const char *)*typeface*

Returns a Font object whose traits are the same as those of *fontObj* except as specified by *typeface*. If the conversion can't be made, the method returns *fontObj* itself. This method can be used to convert a font, or it can be overridden to convert fonts in a different manner.

See also: – **convert:toFamily:**, – **convert:toSize:**, – **convert:toHaveTrait:**, – **convert:toNotHaveTrait:**, – **convertWeight:of:**

convert:toFamily:

– **convert:***fontObj* **toFamily:**(const char *)*family*

Returns a Font object whose traits are the same as those of *fontObj* except as specified by *family*. If the conversion can't be made, the method returns *fontObj* itself. This method can be used to convert a font, or it can be overridden to convert fonts in a different manner.

See also: – **convert:toFace:**, – **convert:toSize:**, – **convert:toHaveTrait:**, – **convert:toNotHaveTrait:**, – **convertWeight:of:**

convert:toHaveTrait:

– **convert:***fontObj* **toHaveTrait:**(NXFontTraitMask)*traits*

Returns a Font object whose traits are the same as those of *fontObj* except as altered by the addition of the traits specified by *traits*. Of course, conflicting traits (such as NX_CONDENSED and NX_EXPANDED) have the effect of turning each other off. If the conversion can't be made, the method returns *fontObj* itself. This method can be overridden to convert fonts in a different manner.

See also: – **convert:toNotHaveTrait:**, – **convert:toFace:**, – **convert:toSize:**, – **convert:toFamily:**, – **convertWeight:of:**

convert:toNotHaveTrait:

– **convert:***fontObj* **toNotHaveTrait:**(NXFontTraitMask)*traits*

Returns a Font object whose traits are the same as those of *fontObj* except as altered by the removal of the traits specified by *traits*. If the conversion can't be made, the method returns *fontObj* itself. This method can be overridden to convert fonts in a different manner.

See also: – **convert:toHaveTrait:**, – **convert:toFace:**, – **convert:toSize:**, – **convert:toFamily:**, – **convertWeight:of:**

convert:toSize:

– **convert:***fontObj* **toSize:**(float)*size*

Returns a Font object whose traits are the same as those of *fontObj* except as specified by *size*. If the conversion can't be made, the method returns *fontObj* itself. This method can be used to convert a font, or it can be overridden to convert fonts in a different manner.

See also: – **convert:toFace:**, – **convert:toFamily:**, – **convert:toHaveTrait:**, – **convert:toNotHaveTrait:**, – **convertWeight:of:**

convertFont:

– **convertFont:***fontObj*

Converts *fontObj* according to the user's selections from the Font panel or menu. Whenever an object receives a **changeFont:** message from the FontManager, it should send a **convertFont:** message for each font in its selection.

This method determines what to do to the *fontObj* by checking the **whatToDo** instance variable and applying the appropriate conversion method. Returns the converted font.

convertWeight:of:

– **convertWeight:**(BOOL)*upFlag* **of:***fontObj*

Attempts to increase (if *upFlag* is YES) or decrease (if *upFlag* is NO) the weight of the font specified by *fontObj*. If it can, it returns a new font object with the higher (or lower) weight. If it can't, it returns *fontObj* itself. By default, this method converts the weight only if it can maintain all of the traits of the original *fontObj*. This method can be overridden to convert fonts in a different manner.

See also: – **convert:toHaveTrait:**, – **convert:toNotHaveTrait:**, – **convert:toFamily:**

delegate

– **delegate**

Returns the FontManager's delegate.

See also: – **setDelegate:**

findFont:traits:weight:size:

– **findFont:**(const char *)*family*
traits:(NXFontTraitMask)*traits*
weight:(int)*weight*
size:(float)*size*

If there's a font on the system with the specified *family*, *traits*, *weight*, and *size*, then it's returned; otherwise, **nil** is returned. If NX_BOLD or NX_UNBOLD is one of the traits, *weight* is ignored.

finishUnarchiving

– **finishUnarchiving**

Finishes the unarchiving task by instantiating the one application-wide instance of the FontManager class if necessary.

getFamily:traits:weight:size:ofFont:

– **getFamily:**(const char **)*family*
traits:(NXFontTraitMask *)*traits*
weight:(int *)*weight*
size:(float *)*size*
ofFont:*fontObj*

For the given font object *fontObj*, copies the font family, traits, weight, and point size information into the storage referred to by this method's arguments.

getFontMenu:

– **getFontMenu:**(BOOL)*create*

Returns a menu suitable for insertion in an application's menu. The menu contains an item that brings up the Font panel as well as some common accelerators (such as Bold and Italic). If the *create* flag is YES, the menu is created if it doesn't already exist.

See also: – **getFontPanel:**

getFontPanel:

– **getFontPanel:**(BOOL)*create*

Returns the FontPanel that will be used when the user chooses the Font Panel command from the Font menu. If the *create* flag is YES, the FontPanel is created if it doesn't already exist.

Unless you've specified a different class (by sending a **setFontPanelFactory:** message to the FontManager class before creating the FontManager object), an object of the FontPanel class is returned.

See also: – **getFontMenu:**

isEnabled

– (BOOL)**isEnabled**

Reports whether the controls in the Font panel and the commands in the Font menu are enabled or disabled.

See also: – **setEnabled:**

isMultiple

– (BOOL)**isMultiple**

Returns whether the currently selected text has multiple fonts.

See also: – **setSelFont:isMultiple:**

modifyFont:

– **modifyFont:***sender*

Causes the FontManager's action message (by default, **changeFont:**) to be sent up the responder chain. When the responder replies with a **convertFont:** message, the font is converted in a way specified by the **selectedTag** of the *sender* of this message. The Larger, Smaller, Heavier, and Lighter commands in the Font menu invoke this method.

See also: – **addFontTrait:**, – **removeFontTrait:**

modifyFontViaPanel:

– **modifyFontViaPanel:***sender*

Causes the FontManager's action message (by default, **changeFont:**) to be sent up the responder chain. When the receiver replies with a **convertFont:** message, the FontManager sends a **panelConvertFont:** message to the FontPanel to complete the conversion.

This message is almost always sent by a Control in the Font panel itself. Usually, the panel uses the FontManager's convert routines to do the conversion based on the choices the user has made.

See also: – **panelConvertFont:** (FontPanel)

orderFrontFontPanel:

– **orderFrontFontPanel:***sender*

Sends **orderFront:** to the FontPanel. If there's no Font panel yet, a **new** message is sent to the FontPanel class object, or to the object you specified with the FontManager's **setFontPanelFactory:** class method.

removeFontTrait:

– **removeFontTrait:***sender*

Causes the FontManager's action message (by default, **changeFont:**) to be sent up the responder chain. When the responder replies with a **convertFont:** message, the font is converted to remove the trait specified by *sender.*

Before the action message is sent up the responder chain, the FontManager sets its **traitToChange** variable to the value returned by sending *sender* a **selectedTag** message. The FontManager also sets its **whatToDo** variable to NX_REMOVETRAIT. When the **convertFont:** message is received, the FontManager converts the supplied font by sending itself a **convert:toNotHaveTrait:** message.

See also: – **convertFont:**, – **convert:toHaveTrait:**, – **selectedTag** (Control)

selFont

– **selFont**

Returns the last font set with **setSelFont:isMultiple:**.

If you receive a **changeFont:** message from the FontManager and want to find out what font the user has selected from the Font panel, use the following (assuming **theFontManager** is the application's FontManager object):

```
selectedFont = [theFontManager convertFont:[theFontManager selFont]]
```

See also: – **setSelFont:isMultiple:**, – **modifyFont:**

sendAction

– **sendAction**

Sends the FontManager's action message (by default, **changeFont:**) up the responder chain. The sender is always the FontManager object regardless of which user-interface object initiated the sending of the action. The **whatToDo** and possibly **traitToChange** variables should be set appropriately before sending a **sendAction** message.

You rarely, if ever, need to send a **sendAction** message or to override this method. The message is sent by the target/action messages sent by different user-interface objects that allow users to manipulate the font of the current text selection (for example, the Font panel and the Font menu).

See also: – setAction:

setAction:

– **setAction:**(SEL)*aSelector*

Sets the action that's sent when the user selects a new font from the Font panel or from the Font menu. The default is **changeFont:**.

See also: – sendAction

setDelegate:

– **setDelegate:***anObject*

Sets the FontManager's delegate. The delegate can restrict which font names appear in the Font panel.

See also: – delegate

setEnabled:

– **setEnabled:**(BOOL)*flag*

Sets whether the controls in the Font panel and the commands in the Font menu are enabled or disabled. By default, these controls and commands are enabled. Even when disabled, the Font panel allows the user to preview fonts. However, when the Font panel is disabled, the user can't apply the selected font to text in the application's main window.

You can use this method to disable the user interface to the font selection system when its actions would be inappropriate. For example, you might disable the font selection system when your application has no document window.

See also: – isEnabled

setSelFont:isMultiple:

– **setSelFont:***fontObj* **isMultiple:**(BOOL)*flag*

Sets the font that the Font panel is currently manipulating. An object containing a document should send this message every time its selection changes. If the selection contains multiple fonts, *flag* should be YES.

An object shouldn't send this message as part of its handling of a **changeFont:** message, since doing so will cause subsequent **convertFont:** messages to have no effect. This is because if you are converting a font based on what is set in the Font panel and you reset what's in the panel (by sending a **setSelFont:isMultiple:** message), the FontManager can no longer sensibly convert the font since the information necessary to convert it has been lost.

See also: – **selFont**, – **isMultiple**

Methods Implemented by the Delegate

fontManager:willIncludeFont:

– (BOOL)**fontManager:***sender* **willIncludeFont:**(const char *)*fontName*

Responds to a message informing the FontManager's delegate that the FontPanel is about to include *fontName* in the list displayed to the user. *fontName* is the name of the font, for example "Helvetica-Narrow-Bold". If this method returns NO, the font isn't added; otherwise, it is.

A delegate that implements this method can receive multiple **fontManager:willIncludeFont:** messages whenever the Font panel needs updating, such as when the user selects a different family name to determine which typefaces are available. For each typeface within that family, the delegate will receive notification. Consequently, your implementation of this method shouldn't take long to execute.

FontPanel

Inherits From: Panel : Window : Responder : Object

Declared In: appkit/FontPanel.h

Class Description

The FontPanel is a user-interface object that displays a list of available fonts, letting the user preview them and change the font used to display text. The actual changes are made through conversion messages sent to the FontManager. There is only one FontPanel object for each application.

In general, you add the facilities of the FontPanel (and of the other components of the font conversion system: the FontManager and the Font menu) to your application through Interface Builder. You do this by dragging a Font menu into one of your application's menus. At runtime, when the user chooses the Font Panel command for the first time, the FontPanel object will be created and hooked into the font conversion system. You can also create (or access) the FontPanel through either of the **new...** methods.

A FontPanel can be customized by adding an additional View object or hierarchy of View objects (see **setAccessoryView:**). If you want the FontManager to instantiate a panel object from some class other than FontPanel, use the FontManager's **setFontPanelFactory:** method.

Instance Variables

id **faces**;
id **families**;
id **preview**;
id **current**;
id **size**;
id **sizes**;
id **manager**;
id **selFont**;
NXFontMetrics ***selMetrics**;
int **curTag**;

```
id accessoryView;
id setButton;
id separator;
id sizeTitle;
char *lastPreview;
struct _fpFlags {
    unsigned int multipleFont:1;
    unsigned int dirty:1;
} fpFlags;
```

faces	The Typeface browser.
families	The Family browser.
preview	The preview field.
current	The current font field.
size	The Size field.
sizes	The Size browser.
manager	The FontManager object.
selFont	The font of the current selection.
selMetrics	The metrics of **selFont**.
curTag	The tag of the currently displayed font.
accessoryView	The application-customized area.
currentBox	The box displaying the current font.
setButton	The Set button.
separator	The line separating buttons from upper part of panel.
sizeTitle	The title over the Size field and Size browser.
lastPreview	The last font previewed.
fpFlags.multipleFont	True if selection has multiple fonts.
fpFlags.dirty	True if panel was updated while not visible.

Method Types

Creating a FontPanel	+ new + newContent:style:backing:buttonMask:defer:

Setting the font	– panelConvertFont: – setPanelFont:isMultiple:
Configuring the FontPanel	– accessoryView – setAccessoryView: – setEnabled: – isEnabled – worksWhenModal
Editing the FontPanel's fields	– textDidGetKeys:isEmpty: – textDidEnd:endChar:
Displaying the FontPanel	– orderWindow:relativeTo:
Resizing the FontPanel	– windowWillResize:toSize:

Class Methods

alloc

Disables the inherited **alloc** method to prevent multiple FontPanels from being created. There's only one FontPanel object for each application; you access it through either of the **new...** methods. Returns an error message.

See also: + new, + newContent:style:backing:buttonMask:defer:

allocFromZone:

Disables the inherited **allocFromZone** method to prevent multiple FontPanels from being created. There's only one FontPanel object for each application; you access it through either of the **new...** methods. Returns an error message.

See also: + new, + newContent:style:backing:buttonMask:defer:

new

+ new

Returns a FontPanel object by invoking the **newContent:style:backing:buttonMask:defer:** method. An application has no more than one Font panel, so this method either returns the previously created object (if it exists) or creates a new one.

See also: + new

newContent:style:backing:buttonMask:defer:

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

Returns a FontPanel object. An application has no more than one Font panel, so this method either returns the previously created object (if it exists) or creates a new one. The arguments are ignored. This is the designated **new...** method of the FontPanel class.

See also: + **new**

Instance Methods

accessoryView

– **accessoryView**

Returns the application-customized View set by **setAccessoryView:**.

See also: – **setAccessoryView:**

isEnabled

– (BOOL)**isEnabled**

Reports whether the Font panel's Set button is enabled.

See also: – **setEnabled:**

orderWindow:relativeTo:

– **orderWindow:**(int)*place* **relativeTo:**(int)*otherWin*

Repositions the panel in the screen list and updates the panel if it was changed while not visible. *place* can be one of:

NX_ABOVE
NX_BELOW
NX_OUT

If it's NX_OUT, the panel 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 Font

Panel is to be placed above or below. If *otherWin* is 0, the panel will be placed above or below all other windows.

See also: – **orderWindow:relativeTo:** (Window),
– **makeKeyAndOrderFront:** (Window)

panelConvertFont:

– **panelConvertFont:***fontObj*

Returns a Font object whose traits are the same as those of *fontObj* except as specified by the users choices in the Font Panel. If the conversion can't be made, the method returns *fontObj* itself. The FontPanel makes the conversion by using the FontManager's methods that convert fonts. A **panelConvertFont:** message is sent by the FontManager whenever it needs to convert a font as a result of user actions in the Font panel.

setAccessoryView:

– **setAccessoryView:***aView*

Customizes the Font panel by adding *aView* above the action buttons at the bottom of the panel. The FontPanel is automatically resized to accommodate *aView.*

aView should be the top View in a view hierarchy. If *aView* is **nil**, any existing accessory view is removed. If *aView* is the same as the current accessory view, this method does nothing. Returns the previous accessory view or **nil** if no accessory view was previously set.

See also: – **accessoryView**

setEnabled:

– **setEnabled:**(BOOL)*flag*

Sets whether the Font panel's Set button is enabled (the default state). Even when disabled, the Font panel allows the user to preview fonts. However, when the Font panel is disabled, the user can't apply the selected font to text in the application's main window.

You can use this method to disable the user interface to the font selection system when its actions would be inappropriate. For example, you might disable the font selection system when your application has no document window.

See also: – **isEnabled**

setPanelFont:isMultiple:

– **setPanelFont:***fontObj* **isMultiple:**(BOOL)*flag*

Sets the font that the FontPanel is currently manipulating. This message should *only* be sent by the FontManager. Do not send a **setPanelFont:isMultiple:** message directly.

textDidEnd:endChar:

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

A **textDidEnd:endChar:** message is sent to the FontPanel object when editing is completed in the Size field. This method updates the Size browser and the preview field.

See also: – **textDidGetKeys:isEmpty:**, – **textDidEnd:endChar:** (Text)

textDidGetKeys:isEmpty:

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

A **textDidGetKeys:isEmpty:** message is sent to the FontPanel object whenever the Size field is typed in or emptied.

See also: – **textDidEnd:endChar:**, – **textDidGetKeys:isEmpty:** (Text)

windowWillResize:toSize:

– **windowWillResize:***sender* **toSize:**(NXSize *)*frameSize*

Keeps the FontPanel from being sized too small to accommodate the browser columns and accessory view.

See also: – **windowWillResize:toSize:** (Window)

worksWhenModal

– (BOOL)**worksWhenModal**

Returns whether the FontPanel will operate while a modal panel is displayed within the application. By default, this method returns YES.

See also: – **worksWhenModal** (Panel)

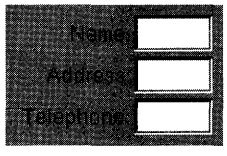
Form

Inherits From: Matrix : Control : View : Responder : Object

Declared In: appkit/Form.h

Class Description

A Form is a Matrix that contains titled entries into which a user can type data values. Here's an example:



Entries are indexed starting with zero at the top. Each item in the Form, including the title, is a FormCell. A mouse click a FormCell (that is, on the title or in the entry area) starts text editing in that entry. If the user presses the Return or Enter key while editing an entry, the action of the entry is sent to the target of the entry, or—if the entry doesn't have an action—the Form sends its action ito its target. If the user presses the Tab key, the next entry in the Form is selected; if the user presses Shift-Tab, the previous entry is selected.

For more information, see the FormCell and Matrix class specifications.

Instance Variables

None declared in this class.

Method Types

Setting Form's Cell class	+ setCellClass:
Initializing a Form	– initFrame:
Laying out the Form	– addEntry: – addEntry:tag:target:action: – insertEntry:at: – insertEntry:at:tag:target:action: – removeEntryAt: – setInterline:
Assigning a tag	– setTag:at:
Finding indices	– findIndexWithTag: – selectedIndex
Modifying graphic attributes	– setBezeled: – setBordered: – setFont: – setTitleFont: – setTextFont: – setTitleAlignment: – setTextAlignment:
Setting item titles	– setTitle:at: – titleAt:
Setting item values	– setDoubleValue:at: – doubleValueAt: – setFloatValue:at: – floatValueAt: – setIntValue:at: – intValueAt: – setStringValue:at: – stringValueAt:
Editing text	– selectTextAt:
Resizing the Form	– calcSize – setEntryWidth: – sizeTo:: – sizeToFit
Displaying	– drawCellAt:
Target and action	– setAction:at: – setTarget:at:

Class Methods

setCellClass:

+ **setCellClass:***classId*

Configures the Form class to use instances of *classId* for its Cells. *classId* should be the **id** of a subclass of FormCell, obtained by sending the **class** message to either the FormCell subclass object or to an instance of that subclass. The default Cell class is FormCell. Returns **self**.

"Creating New Controls" in the Control class specification has more information on how to safely set the Cell class used by a subclass of Control.

See also: – **initFrame:**

Instance Methods

addEntry:

– **addEntry:**(const char *)*title*

Adds a new item with *aString* as the title to the bottom of the receiving Form and returns the FormCell created. The new FormCell has no tag, target, or action, but is enabled and editable. Does not redraw the Form even if autodisplay is on.

See also: – **addEntry:tag:target:action:**

addEntry:tag:target:action:

– **addEntry:**(const char *)*title*
tag:(int)*anInt*
target:*anObject*
action:(SEL)*aSelector*

Adds a new item with *aString* as the title to the bottom of the receiving Form and returns the FormCell created. The FormCell's tag is set to *anInt*, it action to *aSelector*, and its target to *anObject*. The new FormCell is enabled and editable. Does not redraw the Form even if autodisplay is on.

See also: – **addEntry:**

calcSize

– **calcSize**

Calculates the size and layout of the Form based on the sizes of its Cells and their title portions. Your code should invoke this method before drawing if it modifies any of the Cells in the Form in such a way that the size of the Cells or the size of the title part of the Cells has changed. This method is automatically invoked before any drawing is done after a **setTitle:at:**, **setFont:**, **setBezeled:** or some other similar Form method has been invoked.

See also: – **validateSize:** (Matrix)

doubleValueAt:

– (double)**doubleValueAt:**(int)*index*

Returns the value of the entry at position *index* as a double-precision floating point number. Form does not override Control's **doubleValue** method; your code should never use that method with a Form.

See also: – **setDoubleValue:at:**, – **floatValueAt:**, – **intValueAt:**, – **stringValueAt:**, – **doubleValue** (Control)

drawCellAt:

– **drawCellAt:**(int)*index*

Displays the FormCell at the specified *index* in the Form.

findIndexWithTag:

– (int)**findIndexWithTag:**(int)*aTag*

Returns the index of the Cell with the corresponding tag, –1 otherwise.

See also: – **findCellWithTag:** (Matrix)

floatValueAt:

– (float)**floatValueAt:**(int)*index*

Returns the value of the entry at position *index* as a single-precision floating point number. Form does not override Control's **floatValue** method; your code should never use that method with a Form.

See also: – **setFloatValue:at:**, – **doubleValueAt:**, – **intValueAt:**, – **stringValueAt:**, – **floatValue** (Control)

initFrame:

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

Initializes and returns the receiver, a new instance of Form, with default parameters in the given frame. The new Form has no entries. Newly created entries will have the following default characteristics: titles will be right aligned, text will be left justified with bezeled borders, entry background color will be white, text color black, fonts will be the user's chosen system font in 12.0, the interline spacing will be 1.0, and the actions will be NULL. This method is the designated initializer for Form; override it if you create a subclass of Form that performs its own initialization.

Note that Form doesn't override the Matrix class's designated initializers **initFrame:mode:cellClass:numRows:numCols:** or **initFrame:mode:prototype:numRows:numCols:**. Don't use those methods to initialize a new instance of Form.

See also: – **initFrame:** (Matrix), – **initFrame:mode:cellClass:numRows:numCols:**, – **initFrame:mode:prototype:numRows:numCols:**

insertEntry:at:

– **insertEntry:**(const char *)*title* **at:**(int)*index*

Inserts an item with the title *title* at position *index* in the Form. The item at the top of the form has an index of 0. The new FormCell has no tag, target, or action. Returns the FormCell used to implement the entry. Returns the newly inserted FormCell. Does not redraw the Form even if autodisplay is on.

See also: – **insertEntry:at:tag:target:action:**, – **addEntry:**, – **addEntry:tag:target:action:**, – **removeEntryAt:**

insertEntry:at:tag:target:action:

– **insertEntry:**(const char *)*title*
at:(int)*index*
tag:(int)*anInt*
target:*anObject*
action:(SEL)*aSelector*

Inserts a new entry with the given *title* at position *index*. The tag, target, and action of the corresponding entry are set to the given values. Returns the newly inserted FormCell. Does not redraw the Form even if autodisplay is on.

See also: – **insertEntry:at:**, – **addEntry:**, – **addEntry:tag:target:action:**, – **removeEntryAt:**

intValueAt:

– (int)**intValueAt:**(int)*index*

Returns the value of the entry at position *index* as an integer. Form does not override Control's **intValue** method; your code should never use that method with a Form.

See also: – **setIntValue:at:**, – **doubleValueAt:**, – **floatValueAt:**, – **stringValueAt:**, – **intValue** (Control)

removeEntryAt:

– **removeEntryAt:**(int)*index*

If *index* is a valid position in the Form, removes the entry there and frees it. Note that if you use Matrix's **removeRowAt:andFree:** method to remove an entry, the widths of the titles in the entries will not be readjusted; your code should use this method instead. Does not redraw the Form even if autodisplay is on. Returns **self**.

See also: – **addEntry:**, – **insertEntry:at:**

selectTextAt:

– **selectTextAt:**(int)*index*

If *index* is a valid position in the Form, begins text editing on the item at that position. Returns the FormCell selected.

selectedIndex

– (int)**selectedIndex**

Returns the index of the currently selected entry if any, –1 otherwise. The currently selected entry is the one being edited or, if none of the entries is being edited, then it's the entry that was last edited.

setAction:at:

– **setAction:**(SEL)*aSelector* **at:**(int)*index*

Sets the action of the FormCell at position *index* to *aSelector*. Returns **self**.

See also: – **action** (ActionCell), – **setTarget:at:**

setBezeled:

– **setBezeled:**(BOOL)*flag*

If *flag* is YES, all Cells in the Form are set to show a bezel around their editable text and are redrawn; if *flag* is NO, Cells in the Form have no bezel. A bezel is mutually exclusive with a border, and invoking this method with NO as the argument will not remove a border. Returns **self**.

See also: – **isBezeled** (Cell), – **setBordered:**

setBordered:

– **setBordered:**(BOOL)*flag*

If *flag* is YES, all Cells in the Form are set to show a 1-pixel black border around their editable text and are redrawn; if *flag* is NO, Cells in the Form have no border. A border is mutually exclusive with a bezel, and invoking this method with NO as the argument will not remove a bezel. Returns **self**.

See also: – **isBordered** (Cell), – **setBezeled:**

setDoubleValue:at:

– **setDoubleValue:**(double)*aDouble* **at:**(int)*index*

Sets the value of the item at position *index* to *aDouble* and redraws that item. Form does not override Control's **setDoubleValue:** method; your code should never use that method with a Form.

See also: – **doubleValueAt:**, – **setFloatValue:at:**, – **setIntValue:at:**, – **setStringValue:at:**, – **setDoubleValue:** (Control)

setEntryWidth:

– **setEntryWidth:**(NXCoord)*width*

Sets the width of all the entries (including the title part). Doesn't redisplay the Form. You should invoke **sizeToFit** after invoking this method. Returns **self**.

See also: – **sizeToFit**

setFloatValue:at:

– **setFloatValue:**(float)*aFloat* **at:**(int)*index*

Sets the value of the item at position *index* to *aFloat* and redraws that item. Form does not override Control's **setFloatValue:** method; your code should never use that method with a Form.

See also: – **floatValueAt:**, – **setDoubleValue:at:**, – **setIntValue:at:**, – **setStringValue:at:**, – **setFloatValue:** (Control)

setFont:

– **setFont:***fontObject*

Sets the Font used to draw both the titles and the editable text in the Form. It's generally best to keep the title Font and the text Font the same (or at least the same size); therefore, this method is preferred to **setTitleFont:** and **setTextFont:**. Redraws the Form if autodisplay is on. Returns **self**.

See also: – **setTitleFont:**, – **setTextFont:**

setIntValue:at:

– **setIntValue:**(int)*anInt* **at:**(int)*index*

Sets the value of the item at position *index* to *anInt* and redraws that item. Form does not override Control's **setIntValue:** method; your code should never use that method with a Form.

See also: – **intValueAt:**, – **setDoubleValue:at:**, – **setFloatValue:at:**, – **setStringValue:at:**, – **setIntValue:** (Control)

setInterline:

– **setInterline:**(NXCoord)*spacing*

Sets the space between items in the Form to *spacing*. Does not redraw the matrix even if autodisplay is on. Returns **self**.

setStringValue:at:

– **setStringValue:**(const char *)*aString* **at:**(int)*index*

Sets the value of the item at position *index* to *aString* and redraws that item. Form does not override Control's **setStringValue:** method; your code should never use that method with a Form.

See also: – **stringValueAt:**, – **setFloatValue:at:**, – **setDoubleValue:at:**, – **setStringValue:at:**, – **setStringValue:** (Control)

setTag:at:

– **setTag:**(int)*anInt* **at:**(int)*index*

Sets the tag of the FormCell at position *index* to *anInt*. Returns **self**.

See also: – **tag** (ActionCell)

setTarget:at:

– **setTarget:***anObject* **at:**(int)*index*

Sets the target of the FormCell at position *index* to *anObject*. Returns **self**.

See also: – **target** (ActionCell), – **setAction:at:**

setTextAlignment:

– **setTextAlignment:**(int)*mode*

Sets the alignment mode for the editable text in the Form. *mode* can be one of three constants: NX_LEFTALIGNED, NX_CENTERED, or NX_RIGHTALIGNED. The default is left aligned. Redraws the Form if autodisplay is on, and returns **self**.

See also: – **setTitleAlignment:**

setTextFont:

– **setTextFont:***fontObject*

Sets the Font used to draw the editable text in the Form to *fontObject*. Redraws the Form if autodisplay is on, and returns **self**.

See also: – **setFont:**, – **setTitleFont:**

setTitle:at:

– **setTitle:**(const char *)*aString* **at:**(int)*index*

Changes the title of the entry at position *index* to *aString*.

See also: – **titleAt:**

setTitleAlignment:

– **setTitleAlignment:**(int)*mode*

Sets the alignment mode for titles in the Form. *mode* can be one of three constants: NX_LEFTALIGNED, NX_CENTERED, or NX_RIGHTALIGNED. The default is right aligned. Redraws the Form if autodisplay is on, and returns **self**.

See also: – **setTextAlignment:**

setTitleFont:

– **setTitleFont:***fontObject*

Sets the Font used to draw the titles in the Form. to *fontObject* Redraws the Form if autodisplay is on, and returns **self**.

See also: – **setFont:**, – **setTextFont:**

sizeTo::

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

Resizes the entry width to reflect *width*, then resizes the Form to *width* and *height*. Doesn't redraw the Form. Returns **self**.

See also: – **sizeToFit**

sizeToFit

– **sizeToFit**

Adjusts the width of the Form so that it's the same as the width of the entries. Adjusts the height of the Form so that it will exactly contain all the Cells. Doesn't redraw the Form. Returns **self**.

See also: – **sizeTo::**, – **setEntryWidth:**

stringValueAt:

– (const char *)**stringValueAt:**(int)*index*

Returns the value of the entry at position *index* as a string. Form does not override Control's **stringValue** method; your code should never use that method with a Form.

See also: – **setStringValue:at:**, – **doubleValueAt:**, – **floatValueAt:**, – **intValueAt:**, – **stringValue** (Control)

titleAt:

– (const char *)**titleAt:**(int)*index*

Returns the title of the entry at position *index*.

See also: – **setTitle:at:**

FormCell

Inherits From: ActionCell : Cell : Object

Declared In: appkit/FormCell.h

Class Description

This class is used to implement entries in a Form. It displays a title within itself, and allows editing only in the remaining (right-hand) portion of the Cell.

See the Form class specification for more on the use of FormCell.

Instance Variables

NXCoord **titleWidth**;
id **titleCell**;
NXCoord **titleEndPoint**;

titleWidth	The width of the title portion; if –1, width is calculated as needed.
titleCell	The Cell used to draw the title.
titleEndPoint	The coordinate that separates the title from the text area.

Method Types

Initializing, copying, and freeing a FormCell	– init – initTextCell: – copyFromZone: – free
Determining a FormCell's size	– calcCellSize:inRect:
Enabling the FormCell	– setEnabled:

Modifying the title	– setTitle: – title – setTitleFont: – titleFont – setTitleAlignment: – titleAlignment – setTitleWidth: – titleWidth: – titleWidth
Modifying graphic attributes	– isOpaque
Displaying	– drawInside:inView: – drawSelf:inView:
Managing cursor rectangles	– resetCursorRect:inView:
Tracking the mouse	– trackMouse:inRect:ofView:
Archiving	– read: – write:

Instance Methods

calcCellSize:inRect:

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

Calculates the size of the FormCell assuming it's constrained to fit within *aRect*. Returns the size in *theSize*.

copyFromZone:

– **copyFromZone:**(NXZone *)*zone*

Creates and returns a copy of the receiving FormCell instance allocated from *zone*.

drawInside:inView:

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

Draws only the text inside the FormCell (not the bezel or the title of the FormCell). If you create a subclass of FormCell and override **drawSelf:inView:**, you must implement this method as well. Returns **self**.

See also: – **drawSelf:inView:**

drawSelf:inView:

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

Has the FormCell's title Cell drawn, then draws the editable text portion of the FormCell. returns **self**.

See also: – **drawInside:inView:**

free

– **free**

Frees the storage used by the FormCell and returns **nil**.

init

– **init**

Initializes and returns the receiver, a new instance of FormCell, with its contents set to an empty string ("") and its title set to "Field", right-aligned.

See also: – **initTextCell:**

initTextCell:

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

Initializes and returns the receiver, a new instance of FormCell, with its contents set to the empty string ("") and its title set to *aString*. The font for both title and text is the user's chosen system font in 12.0 point, and the text area is drawn with a bezel. This method is the designated initializer for FormCell.

See also: – **init**

isOpaque

– (BOOL)**isOpaque**

Returns YES if the FormCell is opaque, NO otherwise. If the FormCell has a title, then it's not opaque (since the title field is not opaque).

See also: – **isOpaque** (Cell)

read:

– **read:**(NXTypedStream *)*stream*

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

See also: – **write:**

resetCursorRect:inView:

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

Adds a cursor rectangle to *controlView* (with **addCursorRect:cursor:**), allowing the cursor to change to an I-beam when it passes over the text portion of the FormCell.

See also: – **addCursorRect:cursor:** (View, Control)

setEnabled:

– **setEnabled:**(BOOL)*flag*

If *flag* is YES, the FormCell accepts mouse clicks; if NO, it doesn't.

See also: – **isEnabled** (Cell)

setTitle:

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

Sets the title of the FormCell to *aString*.

See also: – **title**

setTitleAlignment:

– **setTitleAlignment:**(int)*mode*

Sets the alignment of the title. *mode* can be one of three constants: NX_LEFTALIGNED, NX_CENTERED, or NX_RIGHTALIGNED.

See also: – **titleAlignment**

setTitleFont:

– **setTitleFont:***fontObject*

Sets the Font used to draw the title of the FormCell.

See also: – **setFont:**

setTitleWidth:

– **setTitleWidth:**(NXCoord)*width*

Sets the width of the title field to *width*. If *width* is –1, the title field's width is always calculated when needed. Use this method only if the FormCell's title isn't going to change, or if your code always resets the title width when it resets the title.

See also: – **titleWidth**, – **titleWidth:**

title

– (const char *)**title**

Returns the title of the FormCell.

See also: – **setTitle:**

titleAlignment

– (int)**titleAlignment**

Returns the alignment of the title, which will be one of the following: NX_LEFTALIGNED, NX_CENTERED, or NX_RIGHTALIGNED.

See also: – **setTitleAlignment:**

titleFont

– **titleFont**

Returns the Font used to draw the title of the FormCell.

See also: – **setTitleFont:**

titleWidth

– (NXCoord)**titleWidth**

If the width of the title has already been set, then that value is returned. Otherwise, it's calculated and returned.

See also: – **setTitleWidth:**, – **titleWidth:**

titleWidth:

– (NXCoord)**titleWidth:**(const NXSize *)*aSize*

If the title width has been set, then it's returned. Otherwise, the width is calculated constrained to *aSize*. *aSize* may be NULL, in which case the width is calculated without constraint.

See also: – **setTitleWidth:**, – **titleWidth:**

trackMouse:inRect:ofView:

– (BOOL)**trackMouse:**(NXEvent*)*event*
inRect:(const NXRect*)*aRect*
ofView:*controlView*

Causes editing to occur. Returns YES if the mouse goes up in the FormCell, NO otherwise.

See also: – **trackMouse:inRect:ofView:** (TextFieldCell)

write:

– **write:**(NXTypedStream *)*stream*

Writes the receiving FormCell to the typed stream *stream*. Returns **self**.

See also: – **read:**

Listener

Inherits From: Object

Declared In: appkit/Listener.h

Class Description

The Listener class, with the Speaker class, supports communication between applications through Mach messaging. Mach messages are the standard way of performing remote procedure calls (RPCs) in the Mach operating system. The Listener class implements the receiving end of a remote message, and the Speaker class implements the sending end.

Remote messages are sent to ports, which act something like mailboxes for the tasks that have the right to receive the messages delivered there. Each Listener corresponds to a single Mach port to which its application has receive rights. Since a port has a fixed size—usually there's room for only five messages in the port queue—when the port is full, a new message must wait for the Listener to take an old message from the queue.

To initiate a remote message, you send an Objective C message to a Speaker instance. The Speaker translates it into the proper Mach message protocol and dispatches it to the port of the receiving task. The Mach message is received by the Listener instance associated with the port. The Listener verifies that it understands the message, that the Speaker has sent the correct parameters for the message, and that all data values are well formed—for example, that character strings are null-terminated. The Listener translates the Mach message back into an Objective C message, which it sends to itself. It's as if an Objective C message sent to a Speaker in one task is received by a Listener in another task.

Delegation

The Listener methods that receive remote Objective C messages simply pass those messages on to a delegate. The Listener's job is just to get the message and find another object to respond to it.

The **setDelegate:** method assigns a delegate to the Listener. There's no default delegate, but before the Application object gets its first event, it registers a Listener for the application and makes itself the Listener's delegate. You can register your own Listener (with Application's **setAppListener:** method) in start-up code, and when you send the Application object a **run** message, the Application object will become the Listener's delegate.

If an object has its own delegate when it becomes the Listener's delegate, the Listener looks first to its delegate's delegate and only then to its own delegate when searching for an object to entrust with a remote message. This means that you can implement the methods that respond to remote messages in either the Application object's delegate or in the Application object. (You can also implement the methods directly in a Listener subclass, or in another object you make the Listener's delegate.)

Setting Up a Listener

Two methods, **checkInAs:** and **usePrivatePort**, allocate a port for the Listener:

- With the **checkInAs:** method, the Listener's port is given a name (usually the name of the application) and is registered with the network name server. This makes the port publicly available so that other applications can find it. Applications get send rights to a public port through the **NXPortFromName()** function.
- Alternatively, the Listener's port can be kept private (with the **usePrivatePort** method). Send rights to the port can then be doled out only to selected applications.

Once allocated, the port must be added (with the **addPort** method) to the list of those that the client library monitors. A procedure will automatically be called to read Mach messages from the port queue and begin the Listener's process of transforming the Mach message back into an Objective C message. The procedure is called between events, provided the priority of getting remote messages is at least as high as the priority of getting the next event.

A Listener is typically set up as follows:

```
myListener = [[Listener alloc] init];
[myListener setDelegate:someOtherObject];
            /*
             * Sets the object responsible for handling
             * messages received.
             */
[myListener checkInAs:"portname"];
            /* or [myListener usePrivatePort] */
[myListener addPort];
            /*
             * Now, between events, the client library
             * will check to see if a message has arrived
             * in the port queue.
             */
. . .
[myListener free];
            /* When we no longer need the Listener. */
```

An application may have more than one Listener and Speaker, but it must have at least one of each to communicate with the Workspace Manager and other applications. If your application doesn't create them, a default Listener and Speaker are created for you at start-up before Application's **run** method gets the first event.

If a Listener is created for you, it will be checked in automatically under the name returned by Application's **appListenerPortName** method. Normally, this is the name assigned to the application at compile time. The port will also be added to the list of those the client library monitors, so the Listener will be scheduled to receive messages asynchronously.

Remote Methods

The Listener and Speaker classes implement a number of methods that can be used to send and receive remote messages. You can add other methods in Listener and Speaker subclasses. The **msgwrap** program can be used to generate subclass definitions from a list of method declarations. Most programmers will use **msgwrap** instead of manually subclassing the Listener class. See the man page for **msgwrap** for details.

Some remote methods, especially those with the prefix "msg", are designed to allow an application to run under program control rather than user control. By implementing these methods, you'll permit a controlling application to run your application in conjunction with others as part of a script.

Argument Types

Remote messages take two kinds of arguments—input arguments, which pass values from the Speaker to the Listener, and output arguments, which are used to pass values back from the Listener to the Speaker. The Listener sends return information back to the Speaker in a separate Mach message to a port provided by the Speaker. The Speaker reformats this information so that it's returned by reference in variables specified in the original Objective C message.

A method can take up to NX_MAXMSGPARAMS arguments. Arguments are constrained to a limited set of permissible types. Internally, the Listener and Speaker identify each permitted type with a unique character code. Input argument types and their identifying codes are listed below. Note that an array of bytes counts as a single argument, even though two Objective C parameters are used to refer to it—a pointer to the array and an integer that counts the number of bytes in the array. A character string must be null-terminated.

Category	Type	Character Code
integer	(int)	i
double	(double)	d
character string	(char *)	c
byte array	(char *), (int)	b
receive rights (port)	(port_t)	r
send rights (port)	(port_t)	s

There's a matching output argument for each of these categories. Since output arguments return information by reference, they're declared as pointers to the respective input types:

Category	Type	Character Code
integer	(int *)	I
double	(double *)	D
character string	(char **)	C
byte array	(char **), (int *)	B
receive rights (port)	(port_t *)	R
send rights (port)	(port_t *)	S

The validity of all input parameters is guaranteed for the duration of the remote message. The memory allocated for a character string or a byte array is freed automatically after the Listener method returns. If you want to save a string or an array, you must copy it. When the amount of input data is large, you can use the **NXCopyInputData()** function to take advantage of the out-of-line data feature of Mach messaging. This function is passed the index of the 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.

The application is responsible for deallocating all port parameters received with the **port_deallocate()** function when they're no longer needed.

Return Values

All remote methods return an **int** that indicates whether or not the message was successfully transmitted. A return of 0 indicates success.

The Listener methods that receive remote messages use the return value to signal whether they're able to delegate a message to another object. If a method can't entrust its message to the delegate (or the delegate's delegate), it returns a value other than 0. If, on the other hand, it's successful in delegating the message, it passes on the delegate's return value as its own. In general, delegate methods should always return 0.

The Listener doesn't pass the return value back to the Speaker that initiated the remote message. However, if the Speaker is expecting return information from the Listener—that is, if the remote message has output arguments—a nonzero return causes the Listener to send an immediate message back to the Speaker indicating its failure to find a delegate for the remote message. The Speaker method then returns –1.

Note that the return value indicates only whether the message got through; it doesn't say anything about whether the action requested by the message was successfully carried out. To provide that information, a remote message must include an output argument.

Instance Variables

char ***portName**;
port_t **listenPort**;
port_t **signaturePort**;
id **delegate**;
int **timeout**;
int **priority**;

portName	The name under which the port is registered.
listenPort	The port where the Listener receives remote messages.
signaturePort	The port used to authenticate registration.
delegate	The object responsible for responding to remote messages received by the Listener.
timeout	How long, in milliseconds, that the Listener will wait for its return results to be placed in the port queue of the sending application.
priority	The priority level at which the Listener will receive messages.

Method Types

Initializing the class	+ initialize
Initializing a new Listener instance	– init
Freeing a Listener	– free
Setting up a Listener	– addPort – removePort – checkInAs: – usePrivatePort – checkOut – listenPort – signaturePort – portName – setPriority: – priority – setTimeout: – timeout + run

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:
Receiving remote messages	– messageReceived: – performRemoteMethod:paramList: – remoteMethodFor:
Assigning a delegate	– setDelegate: – delegate – setServicesDelegate: – servicesDelegate
Archiving	– read: – write:

Class Methods

initialize

+ initialize

Sets up a table that instances of the class use to recognize the remote messages they understand. The table lists the methods that can receive remote messages and specifies the number of parameters for each along with their types. An **initialize** message is sent to the class the first time it's used; you should never invoke this method.

run

+ run

Sets up the necessary conditions for Listener objects to receive remote messages if they're used in applications that don't have an Application object and a main event loop. In other words, if an application doesn't send a **run** message to the Application object,

```
[NXApp run];
```

it will need to send a **run** message to the Listener class

```
[Listener run];
```

for instances of the class to work. This method never returns, so your application will probably need to be dispatched by messages to its Listener instances.

Instance Methods

addPort

– **addPort**

Enables the Listener to receive messages by adding its port to the list of those that the client library monitors. The Listener will then be scheduled to receive messages between events. Returns **self**.

See also: – **removePort**, **DPSAddPort()**

checkInAs:

– (int)**checkInAs:**(const char *)*name*

Allocates a port for the Listener, and registers that port as *name* with the Mach network name server. This method also allocates a signature port that's used to protect the right to remove *name* from the name server. This method returns 0 if it successfully checks in the application with the name server, and a Mach error code if it doesn't. The Mach error code is most likely to be one of those defined in the header files **servers/netname_defs.h** and **mach/kern_return.h**

See also: – **usePrivatePort**, – **checkOut**

checkOut

– (int)**checkOut**

Removes the Listener's port from the list of those registered with the network name server. This makes the port private. This method will always be successful and therefore always returns 0.

See also: – **checkInAs:**

delegate

– delegate

Returns the Listener's delegate. The default delegate is **nil**, but just before the first event is received, the Application object is made the delegate of the Listener registered as the Application object's Listener. The delegate is expected to respond to the remote messages received by the Listener, although it may do this by sending messages to another object.

See also: – setDelegate:, **– setAppListener:** (Application)

free

– free

Frees the Listener object and deallocates its listen port and its signature port. If the Listener's port is registered with the network name server, it is unregistered.

init

– init

Initializes a newly allocated Listener instance. The new instance has no port name, its priority is set to NX_BASETHRESHOLD, its timeout is initialized to 30,000 milliseconds, its listen port and signature port are both PORT_NULL, and it has no delegate. Returns **self**.

See also: – setPriority:, **– setTimeout:**, **– setDelegate:**, **– checkInAs:**

listenPort

– (port_t)**listenPort**

Returns the port at which the Listener receives remote messages. This port is never set directly, but is allocated by either **checkInAs:** or **usePrivatePort**. It's deallocated by the **free** method. The Listener caches this port as its **listenPort** instance variable.

See also: – checkInAs:, **– usePrivatePort**

messageReceived:

– **messageReceived:**(NXMessage *)*msg*

Begins the process of translating a Mach message received at the Listener's port into an Objective C message. This method verifies that the Mach message is well formed, that it corresponds to an Objective C method understood by the Listener, and that the method's arguments agree in number and type with the fields of the Mach message.

messageReceived: messages are initiated whenever a Mach message is to be read from the Listener's port; you shouldn't initiate them in the code you write. Returns **self**.

See also: – **performRemoteMethod:paramList:**

msgCalc:

– (int)**msgCalc:**(int *)*flag*

Receives a remote message to perform any calculations that are necessary to bring the current window up to date. The method you implement to respond to this message should set the integer specified by *flag* 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*

Receives a remote message requesting the application to copy the current selection to the pasteboard as *aType* data. *aType* should be one of the standard pasteboard types defined in **appkit/Pasteboard.h**. The method you implement to respond to this request should set the integer referred to by *flag* to YES if the selection is copied, and to NO if it isn't.

msgCutAsType:ok:

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

Receives a remote message requesting the application to delete the current selection and place it in the pasteboard as *aType* data. *aType* should be one of the standard pasteboard types defined in **appkit/Pasteboard.h**. The method you implement to respond to this request should set the integer referred to by *flag* to YES if the requested action is carried out, and to NO if it isn't.

msgDirectory:ok:

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

Receives a remote message asking for the current directory. The method you implement to respond to this message should place a pointer to the full path of its current directory in the variable specified by *fullPath*. The integer specified by *flag* should be set to YES if the directory will be provided, and to NO if it won't.

The current directory is application-specific, but is probably best described as the directory the application would show in its Open panel were the user to bring it up.

msgFile:ok:

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

Receives a remote message requesting the application to provide the full pathname of its current document. The current document is the file displayed in the main window.

The method you implement to respond to this request should set the pointer referred to by *fullPath* so that it points to a string containing the full pathname of the current document. The integer specified by *flag* should be set to YES if the pathname is provided, and to NO if it isn't.

msgPaste:

– (int)**msgPaste:**(int *)*flag*

Receives a remote message requesting the application to replace the current selection with the contents of the pasteboard, just as if the user had chosen the Paste command from the Edit menu. The method you implement to respond to this message should set the integer referred to by *flag* to YES if the request is carried out, and to NO if it isn't.

msgPosition:posType:ok:

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

Receives a remote message requesting a description of the current selection.

The method you implement to respond to this request should describe the selection in a character string and set the pointer referred to by *aString* so that it points the description.

The integer referred to by *anInt* should be set to one of the following constants to indicate how the current selection is described:

NX_TEXTPOSTYPE	As a character string to search for
NX_REGEXPRPOSTYPE	As a regular expression to search for
NX_LINENUMPOSTYPE	As a colon-separated range of line numbers, for example "10:12"
NX_CHARNUMPOSTYPE	As a colon-separated range of character positions, for example "21:33"
NX_APPPOSTYPE	As an application-specific description

The integer referred to by *flag* should be set to YES if the requested information is provided in the other two output arguments, and to NO if it isn't.

msgPrint:ok:

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

Receives a remote message requesting the application to print the document whose path is *fullPath*. The method you implement to respond to this request should set the integer referred to by *flag* to YES if the document is printed, and to NO if it isn't. The document file should be closed after it's printed.

msgQuit:

– (int)**msgQuit:**(int *)*flag*

Receives a remote message for the application to quit. The method you implement to respond to this message should set the integer specified by *flag* to YES if the application will quit, and to NO if it won't.

msgSelection:length:asType:ok:

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

Receives a remote message asking the application for its current selection as *aType* data. *aType* will be one of the following standard data types for the pasteboard (or an application-specific type):

NXAsciiPboardType
NXPostScriptPboardType
NXTIFFPboardType
NXRTFPboardType
NXSoundPboardType
NXFilenamePboardType
NXTabularTextPboardType

The method you implement to respond to this request should set the pointer referred to by *bytes* so that it points to the selection and also place the number of bytes in the selection in the integer referred to by *numBytes*. The integer referred to by *flag* should be set to YES if the selection is provided, and to NO if it's not.

msgSetPosition:posType:andSelect:ok:

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

Receives a remote message requesting the application to scroll the current document (the one displayed in the main window) so that the portion described by *aString* is visible. *aString* should be interpreted according to the *anInt* constant, which will be one of the following:

NX_TEXTPOSTYPE	*aString* is a character string to search for.
NX_REGEXPRPOSTYPE	*aString* is a regular expression to search for.
NX_LINENUMPOSTYPE	*aString* is a colon-separated range of line numbers, for example "10:12".
NX_CHARNUMPOSTYPE	*aString* is a colon-separated range of character positions, for example "21:33".
NX_APPPOSTYPE	*aString* is an application-specific description of a portion of the document.

The **msgSetPosition:posType:andSelect:ok:** method you implement should set the integer referred to by *flag* to YES if the document is scrolled, and to NO if it isn't. If *selectFlag* is anything other than 0, the portion of the document described by *aString* should also be selected.

msgVersion:ok:

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

Receives a remote message requesting the current version of the application. The method you implement to respond to this request should set the pointer referred to by *aString* so that it points to a string containing current version information for your application. The integer specified by *flag* should be set to YES if version information is provided, and to NO if it's not.

performRemoteMethod:paramList:

– (int)**performRemoteMethod:**(NXRemoteMethod *)*method*
paramList:(NXParamValue *)*params*

Matches the data received in the Mach message with the corresponding Objective C method and sends the Objective C message to **self**. The Listener method that receives the message will then try to delegate it to another object. *method* is a pointer to the method structure returned by **remoteMethodFor:** and *params* is a pointer to the list of arguments.

The **msgwrap** program automatically generates a **performRemoteMethod:paramList:** method for a Listener subclass. Each Listener subclass must define its own version of the method.

performRemoteMethod:paramList: messages are initiated when the Listener reads a Mach message from its port queue.

See also: – **remoteMethodFor:**, **msgwrap**(8) UNIX manual page

portName

– (const char *)**portName**

Returns the name under which the Listener's port (the port returned by the **listenPort** method) is registered with the network name server.

See also: – **checkInAs:**, – **listenPort**, – **appListenerPortName** (Application)

priority

– (int)**priority**

Returns the priority level for receiving remote messages.

See also: – **setPriority:**

read:

– **read:**(NXTypedStream *)*stream*

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

See also: – **write:**

remoteMethodFor:

– (NXRemoteMethod *)**remoteMethodFor:**(SEL)*aSelector*

Looks up *aSelector* in the table of remote messages the Listener understands and returns a pointer to the table entry. A NULL pointer is returned if *aSelector* isn't in the table.

Each Listener subclass must define its own version of this method and send a message to **super** to perform the Listener version. The **msgwrap** program produces subclass method definitions automatically. The version of the method produced by **msgwrap** uses the **NXRemoteMethodFromSel()** function to do the look up.

remoteMethodFor: messages are initiated automatically when the Listener reads a Mach message from its port queue.

See also: – **performRemoteMethod:paramList:**, **msgwrap**(8) UNIX manual page

removePort

– **removePort**

Removes the Listener's port from the list of those that the client library monitors. Remote messages sent to the port will pile up in the port queue until they are explicitly read; they won't be read automatically between events.

See also: – **addPort**

servicesDelegate

– **servicesDelegate**

Returns the Listener's services delegate, the object that will respond to remote messages sent from the Services menus of other applications. The services delegate should contain the methods that a service-providing application uses to provide services to other applications.

See also: – **setServicesDelegate:**

setDelegate:

– **setDelegate:***anObject*

Sets the Listener's delegate to *anObject*. The delegate is expected to respond to the remote messages received by the Listener. However, if *anObject* has a delegate of its own at the time the **setDelegate:** message is sent, the Listener will first check to see if that object can handle a remote message before checking *anObject*. In other words, the Listener recognizes a chain of delegation.

The delegate assigned by this method will be overridden if the Listener is registered as the Application object's **appListener** and the assignment is made before the Application object is sent a **run** message. Before getting the first event, the **run** method makes the Application object the **appListener**'s delegate.

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

setPriority:

– **setPriority:**(int)*level*

Sets the priority for receiving remote messages to *level*. Whenever the application is ready to get another event, the priority level is compared to the threshold at which the application is asking for the next event. For the Listener to be able to receive remote messages from its port queue, the priority level must be at least equal to the event threshold.

Priority values can range from 0 through 30, but three standard values are generally used:

NX_BASETHRESHOLD	1
NX_RUNMODALTHRESHOLD	5
NX_MODALRESPTHRESHOLD	10

These constants are defined in the **appkit/Application.h** header file.

- At a priority equal to NX_BASETHRESHOLD, the Listener will be able to receive messages whenever the application asks for an event in the main event loop, but not during a modal loop associated with an attention panel nor during a modal loop associated with a control such as a button or slider.
- At a priority equal to NX_RUNMODALTHRESHOLD, the Listener will receive remote messages in the main event loop and in the event loop for an attention panel, but not during a control event loop.
- At a priority equal to NX_MODALRESPTHRESHOLD, remote messages are received even during a control event loop.

The default priority level is NX_BASETHRESHOLD.

A new priority takes effect when the Listener receives an **addPort** message. To change the default, you must either set the Listener's priority before sending it an **addPort** message, or you must send it a **removePort** message then another **addPort** message.

See also: – priority, – addPort

setServicesDelegate:

– setServicesDelegate:*anObject*

Registers *anObject* as the object within a service provider that will respond to remote messages. This method returns **self**. As an example, consider an application called **Thinker** that provides a ThinkAboutIt service that ponders the meaning of ASCII text it receives on the pasteboard. **Thinker** would need to have something like the following in the __services section of its __ICON segment in its Mach-O file:

```
Message: thinkMethod
Port: Thinker
Send Type: NXAsciiPboardType
Menu Item: ThinkAboutIt
```

To get this information in your Mach-O file you could put the above text in a file called **services.txt** and then include the following line in your **Makefile.preamble** file:

```
LDFLAGS = -segcreate __ICON __services services.txt
```

Alternatively, if the services the application can provide are not known at compile time, the application can build a services file at run time; see **NXUpdateDynamicServices()**.

Then, in order to provide the ThinkAboutIt service you must implement a **thinkMethod:userData:error:** method in an object which is the services delegate of a Listener which is listening on the Thinker port. (If the application is named "Thinker", then by default NXApp's Listener listens on this port.) Here is an example method that could be used to provide the ThinkAboutIt service:

```
- thinkMethod:(id)pb
    userData:(const char *)userData
    error:(char **)msg
{
    char *data;
    int length;
    char *const *s;  /* We use s to go through types. */
    char *const *types = [pb types];

    for (s = types; *s; s++)
        if (!strcmp(*s, NXAsciiPboardType)) break;
```

```
    if (*s && [pb readType:NXAsciiPboardType
                    data:&data length:&length])
    {
        /* doSomething is your own method... */
        [self doSomething:data :length];
        /* free the memory allocated by readType:... */
        vm_deallocate(task_self(), data, length);
    }
    /* now make msg point to an error string if */
    /* anything went wrong, and return... */
    return self;
}
```

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

setTimeout:

– **setTimeout:**(int)*ms*

Sets, to *ms* milliseconds, how long the Listener will persist in attempting to send a return message back to the Speaker that initiated the remote message. If *ms* is 0, there will be no time limit. The default is 30,000 milliseconds. Returns **self**.

See also: – **timeout**

signaturePort

– (port_t)**signaturePort**

Returns the port that's used to authenticate the Listener's port to the network name server. This port is never set directly, but is allocated by **checkInAs:** and deallocated by **free**.

See also: – **checkInAs:**, – **free**, **netname_check_in()**, **netname_check_out()**

timeout

– (int)**timeout**

Returns the number of milliseconds the Listener will wait for a return message to the Speaker to be successfully placed in the port designated by the Speaker. If it's 0, there's no time limit.

See also: – **setTimeout:**

usePrivatePort

– (int)**usePrivatePort**

Allocates a listening port for the Listener, but doesn't register it publicly. Other tasks can send messages to this Listener only if they are explicitly given the address of the port in a message; the port is not available through the Network Name Server. This method is an alternative to **checkInAs:**. It returns 0 on success and a Mach error code if it can't allocate the port. The error code will be one of those defined in **mach/kern_return.h**.

See also: – **checkInAs:**

write:

– **write:**(NXTypedStream *)*stream*

Writes the Listener to the typed stream *stream*. Returns **self**.

See also: – **read:**

Matrix

Inherits From: Control : View : Responder : Object

Declared In: appkit/Matrix.h

Class Description

Matrix is a class used for creating groups of Cells that work together in various ways. It includes methods for arranging its Cells in rows and columns, either with or without space between them. The only restriction is that all Cells must be the same size. Cells in the Matrix are numbered by row and columns, each starting with 0; for example, the top left Cell would be at (0, 0), and the Cell that's second down and third across would be at (1, 2). A Matrix can have many Cells of different classes, but usually uses only one type of Cell. A Matrix can be set up to create new Cells by copying a prototype Cell, or by allocating and initializing instances of a specific Cell class.

A Matrix adds to Control's target/action paradigm by allowing a separate target and action for each of its Cells in addition to its own, and also by having an action that gets sent when the user double-clicks a Cell, and which is sent in addition to the single-click action. If a Cell doesn't have an action, the Matrix sends its own action to its own target. If a Cell doesn't have a target, the Matrix sends the Cell's action to its own target. The double-click action of a Matrix is always sent to the target of the Matrix.

Since the user might press the mouse button anywhere in the Matrix, and then drag the mouse around, Matrix offers four "selection modes" which determine how Cells behave when the Matrix is tracking the mouse:

NX_TRACKMODE is the most basic mode of operation. All that happens in this mode is that the Cells are asked to track the mouse with **trackMouse:inRect:ofView:** whenever the mouse is inside their bounds. No highlighting is performed. An example of this mode might be a "graphic equalizer" Matrix of sliders. Moving the mouse around causes the sliders to move under the mouse.

NX_HIGHLIGHTMODE is a modification of TRACKMODE. In this mode, a Cell is highlighted before it is asked to track the mouse, then unhighlighted when it is done tracking. This is useful for multiple unconnected Cells which use highlighting to inform the user that they are being tracked (like push-buttons and switches).

NX_RADIOMODE is used when you want no more than one Cell to be selected at a time. It can be used to create a set of buttons of which one and only one is selected (there is the

option of allowing no button to be selected). Any time a Cell is selected, the previously selected Cell is unselected. The canonical example of this mode is a set of radio buttons.

NX_LISTMODE is the opposite of NX_TRACKMODE. Cells are highlighted, but don't track the mouse. This mode can be used to select a range of text values, for example. Matrix supports the standard multiple-selection paradigms of dragging to select, using the shift key to make disjoint selections, and using the alternate key to extend selections.

The best way to learn about selection modes is to play with a Matrix in Interface Builder, testing the Matrix interface with various options and Cell types. You can also create minimal connections to Buttons that play sounds, setting the action to be **performClick:**, which will cause the sounds to be played when you use the Matrix in Interface Builder's test mode.

Instance Variables

id **cellList**;
id **target**;
SEL **action**;
id **selectedCell**;
int **selectedRow**;
int **selectedCol**;
int **numRows**;
int **numCols**;
NXSize **cellSize**;
NXSize **intercell**;
float **backgroundGray**;
float **cellBackgroundGray**;
id **font**;
id **protoCell**;
id **cellClass**;
id **nextText**;
id **previousText**;
SEL **doubleAction**;
SEL **errorAction**;
id **textDelegate**;

```
struct _mFlags {
        unsigned int highlightMode:1;
        unsigned int radioMode:1;
        unsigned int listMode:1;
        unsigned int allowEmptySel:1;
        unsigned int autoscroll:1;
        unsigned int reaction:1;
        unsigned int selectionByRect:1;
} mFlags;
```

cellList	List of the Cells in the Matrix.
target	The object that is sent an action if a Cell doesn't have its own action or its own target.
action	The message sent to the target of the Matrix if a Cell doesn't have its own.
selectedCell	The selected Cell (if there's only one).
selectedRow	The row number of selected Cell.
selectedCol	The column number of selected Cell.
numRows	Number of rows in the Matrix.
numCols	Number of columns in the Matrix.
cellSize	The size of each Cell in the Matrix (they're all the same size).
intercell	Vertical and horizontal spacing between Cells.
backgroundGray	The gray level drawn between Cell in the Matrix.
cellBackgroundGray	The gray level drawn as the background of each Cell.
font	The Font of text in the Cells of the Matrix.
protoCell	A Cell instance copied to make new Cells.
cellClass	The class of Cell used by the Matrix; not used if a prototype is used.
nextText	The object whose text is selected when Tab is pressed.
previousText	The object whose text is selected when Shift-Tab is pressed.
doubleAction	Action sent to the target of the Matrix on a double-click in a Cell.

errorAction	Message sent to the target when a bad value is entered in a text field.
textDelegate	Delegate for Text object delegate methods.
mFlags.highlightMode	True if selection mode is NX_HIGHLIGHTMODE.
mFlags.radioMode	True if selection mode is NX_RADIOMODE.
mFlags.listMode	True if selection mode is NX_LISTMODE.
mFlags.allowEmptySel	True if no selection is allowed in NX_RADIOMODE.
mFlags.autoscroll	True if the Matrix auto-scrolls when in a ScrollView.
mFlags.reaction	True if an action message caused the Cell that triggered the action message to change.
mFlags.selectionByRect	True if a rectangle of Cells in the Matrix can be selected by dragging the cursor.

Method Types

Initializing the Matrix class	+ initialize + setCellClass:
Initializing and freeing a Matrix	– initFrame: – initFrame:mode:cellClass:numRows:numCols: – initFrame:mode:prototype:numRows:numCols: – free
Setting the selection mode	– setMode: – mode
Configuring the Matrix	– setEnabled: – setEmptySelectionEnabled: – isEmptySelectionEnabled – setSelectionByRect: – isSelectionByRect
Setting the Cell class	– setCellClass: – setPrototype: – prototype

Laying out the Matrix	– addCol – addRow – insertColAt: – insertRowAt: – removeColAt:andFree: – removeRowAt:andFree: – makeCellAt:: – putCell:at:: – renewRows:cols: – setCellSize: – getCellSize: – getCellFrame:at:: – setIntercell: – getIntercell: – cellCount – getNumRows:numCols:
Finding Matrix coordinates	– getRow:andCol:ofCell: – getRow:andCol:forPoint:
Modifying individual Cells	– setIcon:at:: – setState:at:: – setTitle:at:: – setTag:at:: – setTag:target:action:at::
Selecting Cells	– selectCell: – selectCellAt:: – selectCellWithTag: – setSelectionFrom:to:anchor:lit: – selectAll: – selectedCell – getSelectedCells: – selectedCol – selectedRow – clearSelectedCell
Finding Cells	– findCellWithTag: – cellAt:: – cellList

Modifying graphic attributes	– setBackgroundColor: – backgroundColor – setBackgroundGray: – backgroundGray – setCellBackgroundColor: – cellBackgroundColor – setCellBackgroundGray: – cellBackgroundGray – setBackgroundTransparent: – isBackgroundTransparent – setCellBackgroundTransparent: – isCellBackgroundTransparent – setFont: – font
Editing text in Cells	– selectText: – selectTextAt::
Setting Tab key behavior	– setNextText: – setPreviousText:
Assigning a Text delegate	– setTextDelegate: – textDelegate
Text object delegate methods	– textWillChange: – textDidChange: – textDidGetKeys:isEmpty: – textWillEnd: – textDidEnd:endChar:
Resizing the Matrix and Cells	– setAutosizeCells: – doesAutosizeCells – calcSize – sizeTo:: – sizeToCells – sizeToFit – validateSize:
Scrolling	– setAutoscroll: – setScrollable: – scrollCellToVisible::

Displaying	– display – drawSelf:: – drawCell: – drawCellAt:: – drawCellInside: – highlightCellAt::lit:
Target and action	– setTarget: – target – setAction: – action – setDoubleAction: – doubleAction – setErrorAction: – errorAction – setTarget:at:: – setAction:at:: – sendAction – sendAction:to: – sendAction:to:forAllCells: – sendDoubleAction – setReaction:
Handling event and action messages	– acceptsFirstMouse – mouseDown: – mouseDownFlags – performKeyEquivalent:
Managing the cursor	– resetCursorRects
Archiving	– read: – write:

Class Methods

initialize

+ **initialize**

Initializes data for the Matrix class object.

setCellClass:

+ **setCellClass:***classId*

Configures the Matrix class to use instances of *classId* for its Cells. *classId* should be the **id** of a subclass of Cell (usually ActionCell), obtained by sending the **class** message to either the Cell subclass object or to an instance of that subclass. The default Cell class is ActionCell. Returns **self**.

Your code should rarely need to invoke this method, since each instance of Matrix can be configured to use its own Cell class (or a prototype that gets copied). The Cell class set with this method is simply a fallback for Matrices initialized with **initFrame:**.

"Creating New Controls" in the Control class specification has more information on how to safely set the Cell class used by a subclass of Control.

See also: – **initFrame:**...

Instance Methods

acceptsFirstMouse

– (BOOL)**acceptsFirstMouse**

Returns NO if the selection mode of the Matrix is NX_LISTMODE, YES if the Matrix is in any other selection mode. The Matrix does not accept first mouse in NX_LISTMODE to prevent the loss of multiple selections.

See also: – **mode**

action

– (SEL)**action**

Returns the default action of the Matrix. The returned method is used when a Cell with no action receives an event which would ordinarily cause its action to be sent—normally a mouse-up in the Cell. In such cases, the Matrix sends its action to its own target.

See also: – **setAction:**, – **target**, – **action** (ActionCell), – **target** (ActionCell)

addCol

– **addCol**

Adds a new column of Cells to the right of the existing columns, creating new Cells if needed with **makeCellAt::**. Does not redraw even if autodisplay is on. Returns **self**.

If the number of rows or columns in the Matrix has been changed with **renewRows:cols:**, then **makeCellAt:** is invoked only if new Cells are needed (since **renewRows:cols:** doesn't free Cells, it just rearranges them). This allows you to grow and shrink a Matrix without repeatedly creating and freeing the Cells.

See also: – **insertColAt:**, – **makeCellAt::**, – **renewRows:cols:**, – **isAutodisplay** (View)

addRow

– **addRow**

Adds a new row of Cells to the bottom of the existing rows, creating new Cells if needed with **makeCellAt::**. Does not redraw even if autodisplay is on. Returns **self**.

If the number of rows or columns in the Matrix has been changed with **renewRows:cols:**, then **makeCellAt:** is invoked only if new Cells are needed (since **renewRows:cols:** doesn't free Cells, it just rearranges them). This allows you to grow and shrink a Matrix without repeatedly creating and freeing the Cells.

See also: – **insertRowAt:**, – **makeCellAt::**, – **renewRows:cols:**, – **isAutodisplay** (View)

backgroundColor

– (NXColor)**backgroundColor**

Returns the color used to draw the background (the space between the Cells).

See also: – **setBackgroundColor:**, – **backgroundGray**, – **cellBackgroundColor**

backgroundGray

– (float)**backgroundGray**

Returns the gray level used to draw the background (the space between the Cells). If the gray level is less than 0, then the background is transparent.

See also: – **setBackgroundGray:**, – **backgroundColor**, – **cellBackgroundGray**

calcSize

– **calcSize**

Your code should never invoke this method. It is invoked automatically by the system if it has to recompute some size information about the Cells. It invokes **calcDrawInfo:** on each Cell in the Matrix. Can be overridden to do more if necessary (Form overrides **calcSize**, for example). Returns **self**.

See also: – **calcSize** (Control, Form), – **validateSize:**

cellAt::

– **cellAt:**(int)*row* **:**(int)*col*

Returns the Cell at row *row* and column *col*, or **nil** if no such Cell exists.

See also: – **getRow:andCol:ofCell:**

cellBackgroundColor

– (NXColor)**cellBackgroundColor**

Returns the color used to fill the background of a Cell.

See also: – **setCellBackgroundColor:**, – **cellBackgroundGray**, – **backgroundColor**

cellBackgroundGray

– (float)**cellBackgroundGray**

Returns the gray value used to fill the background of a Cell before the Cell is drawn. If the gray level is –1.0, then the Cell is transparent.

See also: – **setCellBackgroundGray:**, – **cellBackgroundColor**, – **backgroundGray**

cellCount

– (int)**cellCount**

Returns the number of Cell positions in the Matrix (that is, the number of rows times the number of columns).

See also: – **cellList**

cellList

– **cellList**

Returns a List object that contains the Cells of the Matrix. The Cells in the list are row-ordered; that is, the first row of Cells appear first in the List, then the next row, and so on.

clearSelectedCell

– **clearSelectedCell**

Deselects the selected Cell or Cells, and returns the previously selected Cell (the last of the selected Cells if there were more than one). If the selection mode is NX_RADIOMODE and empty selection is not allowed, this method won't deselect the selected Cell. Doesn't redisplay the Matrix. It's often more convenient to use **selectCellAt::** with a row and column of (–1, –1), since this will clear the selected Cell and redisplay the Matrix.

See also: – **selectCellAt::**, – **mode**, – **setEmptySelectionEnabled:**

display

– **display**

Draws the Matrix. This method invokes **displayFromOpaqueAncestor:::** if any part of the Matrix (either the space between Cells, or any Cell) is transparent, or **display:::** if the entire Matrix is opaque. Returns **self**.

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

doesAutosizeCells

– (BOOL)**doesAutosizeCells**

Returns YES if Cells are resized proportionally to the Matrix when its size changes; the inter-Cell spacing is kept constant. Returns NO if the inter-Cell spacing changes when the Matrix is resized; the Cell size remains constant.

See also: – **setAutosizeCells:**

doubleAction

– (SEL)**doubleAction**

Returns the action sent by the Matrix to its target when the user double-clicks an entry. Unlike NXBrowser, this method returns NULL if there is no double-click action. The double-click action of a Matrix is sent after the appropriate single-click action (for the Cell clicked or for the Matrix if the Cell doesn't have its own action). If there is no double-click action and the Matrix doesn't ignore multiple clicks, the single-click action is sent twice.

See also: – **setDoubleAction:**, – **action**, – **target**, – **sendDoubleAction**, – **ignoreMultiClick:** (Control)

drawCell:

– **drawCell:***aCell*

If *aCell* is in the Matrix, then it's drawn. Does nothing otherwise. Returns **self**. This method is useful for constructs like:

```
[aMatrix drawCell:[[aMatrix cellAt:aRow :aCol] setSomething:anArg]];
```

See also: – **drawCellAt::**, – **drawCellInside:**

drawCellAt::

– **drawCellAt:**(int)*row* **:**(int)*col*

Displays the Cell at (*row*, *col*) if it's in the Matrix. Does nothing otherwise. Returns **self**.

See also: – **drawCell:**, – **drawCellInside:**

drawCellInside:

– **drawCellInside:***aCell*

If *aCell* is in the Matrix, then its inside (usually all but a bezel or border) is drawn.

See also: – **drawCell:**, – **drawCellAt::**, – **drawInside:inView:** (Cell)

drawSelf::

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

Displays the Cells in the Matrix which intersect any of the *rects*.

errorAction

– (SEL)**errorAction**

Returns the action sent to the target of the Matrix when the user enters an illegal value for a Cell's type (as set by Cell's **setEntryType:** method and checked by Cell's **isEntryAcceptable:** method).

See also: – **setErrorAction:**, – **setEntryType:** (Cell), – **isEntryAccectable:** (Cell)

findCellWithTag:

– **findCellWithTag:**(int)*anInt*

Returns the Cell which has a tag matching *anInt*, or **nil** if no such Cell exists in the Matrix.

See also: – **setTag:at::**, – **setTag:** (ActionCell), – **setTag:target:action:at::**, – **selectCellWithTag:**

font

– **font**

Returns the Font used to display text in the Cells of the Matrix, or **nil** if the Cells don't contain text.

See also: – **setFont:**

free

– **free**

Deallocates the storage for the Matrix and all its Cells, and returns **nil**.

getCellFrame:at::

– **getCellFrame:**(NXRect *)*theRect*
at:(int)*row*
:(int)*col*

Returns **self**, and by reference in *theRect* the frame of the Cell that would be drawn at the specified *row* and *col* (whether or not it actually exists).

See also: – **getCellSize:**

getCellSize:

– **getCellSize:**(NXSize *)*theSize*

Returns **self**, and by reference in *theSize* the width and the height of each Cell in the Matrix (all Cells are the same size).

See also: – **getCellFrame:at::**, – **getIntercell:**

getIntercell:

– **getIntercell:**(NXSize *)*theSize*

Returns **self**, and by reference in *theSize* the vertical and horizontal spacing between Cells.

See also: – **getCellSize:**

getNumRows:numCols:

– **getNumRows:**(int *)*rowCount* **numCols:**(int *)*colCount*

Returns **self**, and, by reference in *rowCount* and *colCount*, the number of rows and columns in the Matrix.

getRow:andCol:forPoint:

– **getRow:**(int *)*row*
andCol:(int *)*col*
forPoint:(const NXPoint *)*aPoint*

Returns the Cell at *aPoint* in the Matrix. *aPoint* must be in the coordinate system of the Matrix. If *aPoint* is outside the bounds of the Matrix or in an intercell spacing, **getRow:andCol:forPoint:** returns **nil**. Also returns by reference in *row* and *col* the row and column position of the Cell.

See also: – **getRow:andCol:ofCell:**

getRow:andCol:ofCell:

– **getRow:**(int *)*row*
andCol:(int *)*col*
ofCell:*aCell*

Returns by reference in *row* and *col* the row and column indices for the position of *aCell* within the Matrix. Returns *aCell* if it's in the Matrix, **nil** otherwise.

See also: – **getRow:andCol:forPoint:**

getSelectedCells:

– **getSelectedCells:**(List *)*aList*

Adds to *aList* the Cells of the Matrix that are selected. If *aList* is **nil**, a new List object is created and filled with the selected Cells. Your code may free the List object, but not the Cells in the List. Returns **self**.

See also: – **selectedCell**

highlightCellAt::lit:

– **highlightCellAt:**(int)*row*
:(int)*col*
lit:(BOOL)*flag*

Highlights or unhighlights the Cell at (*row*, *col*) in the Matrix by sending **highlight:inView:lit:** to the Cell. The PostScript focus must be locked on the Matrix when this message is sent. Returns **self**.

See also: – **highlight:inView:lit:** (Cell)

initFrame:

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

Initializes and returns the receiver, a new instance of Matrix, with default parameters in the given frame. The default font is the user's chosen system font in 12.0 point, the default Cell size is 100.0 by 17.0 points, the default inter-Cell spacing is 1.0 point by 1.0 point, the default background gray is –1 (transparent), and the default Cell background gray is also –1. The new Matrix contains no rows or columns. The default mode is NX_RADIOMODE.

See also: – **initFrame:mode:**...

initFrame:mode:cellClass:numRows:numCols:

– **initFrame:**(const NXRect *)*frameRect*
mode:(int)*aMode*
cellClass:*classId*
numRows:(int)*numRows*
numCols:(int)*numCols*

Initializes and returns the receiver, a new instance of Matrix, in *frameRect* with *numRows* rows and *numCols* columns. *aMode* is set as the tracking mode for the Matrix, and can be one of four constants:

NX_TRACKMODE	Just track the mouse inside the Cells
NX_HIGHLIGHTMODE	Highlight the Cell, then track, then unhighlight
NX_RADIOMODE	Allow no more than one selected Cell
NX_LISTMODE	Allow multiple selected Cells

The behavior for these constants is more fully described in the class description. The new Matrix creates and uses Cells of class *classId*, which should be the return value of a **class** message sent to a subclass of Cell.

This method is the designated initializer for Matrices that add Cells by creating instances of a Cell subclass.

See also: – **initFrame:**, – **initFrame:mode:prototype:numRows:numCols:**

initFrame:mode:prototype:numRows:numCols:

– **initFrame:**(const NXRect *)*frameRect*
mode:(int)*aMode*
prototype:*aCell*
numRows:(int)*numRows*
numCols:(int)*numCols*

Initializes and returns the receiver, a new instance of Matrix, in *frameRect* with *numRows* rows and *numCols* columns. *aMode* is set as the tracking mode for the Matrix, and can be one of four constants:

NX_TRACKMODE	Just track the mouse inside the Cells
NX_HIGHLIGHTMODE	Highlight the Cell, then track, then unhighlight
NX_RADIOMODE	Allow no more than one selected Cell
NX_LISTMODE	Allow multiple selected Cells

The behavior for these constants is more fully described in the class description. The new Matrix creates Cells by copying *aCell*, which should be an instance of a subclass of Cell.

This method is the designated initializer for Matrices that add Cells by copying an instance of a Cell subclass.

See also: – **initFrame:**, – **initFrame:mode:cellClass:numRows:numCols:**

insertColAt:

– **insertColAt:**(int)*col*

Inserts a new column of Cells before *col*, creating new Cells with **makeCellAt::**. If *col* is greater than the number of columns in the Matrix, enough columns are created to expand Matrix to be *col* columns wide. This method doesn't redraw even if autodisplay is on. Your code may need to use **sizeToCells** after sending this method to resize the Matrix to fit the newly added Cells. Returns **self**.

If the number of rows or columns in the Matrix has been changed with **renewRows:cols:**, then **makeCellAt:** is invoked only if new Cells are needed (since **renewRows:cols:** doesn't free Cells, it just rearranges them). This allows you to grow and shrink a Matrix without repeatedly creating and freeing the Cells.

See also: – **addCol**, – **insertRowAt:**, – **sizeToCells**, – **makeCellAt::**

insertRowAt:

– **insertRowAt:**(int)*row*

Inserts a new row of Cells before *row*, creating new Cells with **makeCellAt::**. If *row* is greater than the number of rows in the Matrix, enough rows are created to expand Matrix to be *row* rows high. This method doesn't redraw even if autodisplay is on. Your code may need to use **sizeToCells** after sending this method to resize the Matrix to fit the newly added Cells. Returns **self**.

If the number of rows or columns in the Matrix has been changed with **renewRows:cols:**, then **makeCellAt:** is invoked only if new Cells are needed (since **renewRows:cols:** doesn't free Cells, it just rearranges them). This allows you to grow and shrink a Matrix without repeatedly creating and freeing the Cells.

See also: – **addRow**, – **insertColAt:**, – **sizeToCells**, – **makeCellAt::**

isBackgroundTransparent

– (BOOL)**isBackgroundTransparent**

Returns YES if the Matrix background is transparent, NO otherwise.

See also: – **setBackgroundTransparent:**, – **backgroundGray**

isCellBackgroundTransparent

– (BOOL)**isCellBackgroundTransparent**

Returns YES if Cells in the Matrix have transparent backgrounds, NO otherwise.

See also: – **setCellBackgroundTransparent:**, – **cellBackgroundGray**

isEmptySelectionEnabled

– (BOOL)**isEmptySelectionEnabled**

Returns YES if it is possible to have no Cells selected in a radio-mode Matrix, NO otherwise.

See also: – **setEmptySelectionEnabled:**

isSelectionByRect

– (BOOL)**isSelectionByRect**

Returns YES if a rectangle of Cells in the Matrix can be selected by dragging the cursor, NO otherwise.

See also: – **setSelectionFrom:to:anchor:lit:**

makeCellAt::

– **makeCellAt:**(int)*row* **:**(int)*col*

Creates a new Cell at the specified location in the Matrix. If the Matrix has a prototype Cell, it's copied to create the new Cell; if the Matrix has a Cell class set, it allocates and initializes (with **init**) an instance of that class; if the Matrix has not had a Cell class set, the default class, ActionCell, is used. The new Cell's font is set to the font of the Matrix Returns the newly created Cell.

Your code should never invoke this method directly; it's used by **addRow** and other methods when a Cell must be created. It may be overridden to provide more specific initialization of Cells.

See also: – **addCol**, – **addRow**, – **insertColAt:**, – **insertRowAt:**

mode

– (int)**mode**

Returns the selection mode of the Matrix. These modes are explained in the class description.

See also: – **setMode:**, – **initFrame:mode:**...

mouseDown:

– **mouseDown:**(NXEvent *)*theEvent*

Your code should never invoke this method, but you may override it to implement different mouse tracking than Matrix does. The response of the Matrix depends on its selection mode, as explained in the class description.

In any selection mode, a mouse-down in an editable text Cell immediately enters text editing mode. A double-click in any other kind of Cell sends the double-click action of the Matrix (if there is one) in addition to the single-click action.

See also: – **sendAction**, – **sendDoubleAction**

mouseDownFlags

– (int)**mouseDownFlags**

Returns the flags (for example, NX_SHIFTMASK) that were in effect at the mouse-down event that started the current tracking session. Use this method if you want to access these flags, but don't want the overhead of having to use **sendActionOn:** to add NX_MOUSEDOWNMASK to every Cell to get them. This method is valid only during tracking; it's not useful if the target of the Matrix initiates another tracking loop as part of its action method (as a Cell that pops up a PopUpList does, for example).

See also: – **sendActionOn:** (Cell)

performKeyEquivalent:

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

If there is a Cell in the Matrix that has a key equivalent equal to the character in *theEvent*->**data.key.charCode**, that Cell is made to react as if the user had clicked it by highlighting, changing its state as appropriate, sending its action if it has one, and then unhighlighting. Returns YES if a Cell in the Matrix responds to the key equivalent in *theEvent*, NO if no Cell responds.

Your code should never send this message; it is sent when the Matrix or one of its superviews is the first responder and the user presses a key. You may want to override this method to change the way key equivalents are performed or displayed, or to disable them in your subclass.

prototype

– **prototype**

Returns the prototype Cell that is copied whenever a new Cell needs to be made, or **nil** if there is none.

See also: – **setPrototype:**, – **initFrame:mode:prototype:numRows:numCols:**, – **makeCellAt::**

putCell:at::

– **putCell:***newCell*
at:(int)*row*
:(int)*col*

Replaces the Cell at (*row*, *col*) by *newCell*, and returns the old Cell at that position. Draws the new Cell if autodisplay is on.

read:

– **read:**(NXTypedStream *)*stream*

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

See also: – **write:**

removeColAt:andFree:

– **removeColAt:**(int)*col* **andFree:**(BOOL)*flag*

Removes the column at position *col*. If *flag* is YES then the Cells from that column are freed. Doesn't redraw even if autodisplay is on. Your code should normally send **sizeToCells** after invoking this method to resize the Matrix so it fits the reduced Cell count. Returns **self**.

See also: – **removeRowAt:andFree:**, – **addCol**, – **insertColAt:**

removeRowAt:andFree:

– **removeRowAt:**(int)*row* **andFree:**(BOOL)*flag*

Removes the row at position *row*. If *flag* is YES then the Cells from that row are freed. Doesn't redraw even if autodisplay is on. Your code should normally send **sizeToCells** after invoking this method to resize the Matrix so it fits the reduced Cell count. Returns **self**.

See also: – **removeColAt:andFree:**, – **addRow**, – **insertRowAt:**

renewRows:cols:

– **renewRows:**(int)*newRows* **cols:**(int)*newCols*

Changes the number of rows and columns in the Matrix. This method uses the same Cells as before, creating new Cells only if the new size is larger; it never frees Cells. Doesn't display the Matrix even if autodisplay is on. Your code should normally send **sizeToCells** after invoking this method to resize the Matrix so it fits the changed Cell arrangement. This method deselects all Cells in the Matrix. Returns **self**.

See also: – **addRow**, – **addCol**

resetCursorRects

– **resetCursorRects**

Sends **resetCursorRect:inView:** to each Cell in the Matrix. Any Cell that has a cursor rectangle to set up should send the message **addCursorRect:cursor:** back to the Matrix. Returns **self**.

See also: – **resetCursorRect:inView:** (Cell), – **addCursorRect:cursor:** (View)

scrollCellToVisible::

– **scrollCellToVisible:**(int)*row* **:**(int)*col*

If the Matrix is in a scrolling View, then the Matrix will scroll to make the Cell at (*row*, *col*) visible. Returns **self**.

See also: – **scrollRectToVisible:** (View)

selectAll:

– **selectAll:***sender*

If the mode of the Matrix is not NX_RADIOMODE, then all the Cells in the Matrix are selected and highlighted, and the Matrix is redisplayed. The currently selected Cell is unaffected. Editable text Cells are not affected. Returns **self**.

See also: – **selectCell:**, – **selectCellAt::**, – **selectCellWithTag:**, – **selectText:**

selectCell:

– **selectCell:***aCell*

If *aCell* is in the Matrix, then the Cell is selected, the Matrix is redrawn, and the selected Cell is returned. An editable text Cell's text is selected. Returns **nil** if the Cell is not in the Matrix.

See also: – **selectCellAt::**, – **selectCellWithTag:**, – **selectAll:**, – **selectText:**

selectCellAt::

– **selectCellAt:**(int)*row* **:**(int)*col*

Selects the Cell at the position in the Matrix denoted by (*row*, *col*). An editable text Cell's text is selected. If either *row* or *col* is –1, then the current selection is cleared (unless the Matrix is in NX_RADIOMODE and does not allow empty selection). Redraws the affected Cells and returns **self**.

See also: – **selectCell:**, – **selectCellWithTag:**, – **selectAll:**, – **selectText:**

selectCellWithTag:

– **selectCellWithTag:**(int)*anInt*

If the Matrix has a Cell whose tag is equal to *anInt*, that Cell is selected. An editable text Cell's text is selected. Returns **self**, or **nil** if there is no such Cell.

See also: – **selectCell:**, – **selectCellAt::**, – **selectAll:**, – **selectText:**

selectedCell

– **selectedCell**

Returns the currently selected Cell, or **nil** if no Cell is selected. If more than one Cell is selected, returns the last selected Cell; that is, the Cell that is lowest and furthest to the right in the Matrix.

See also: – **getSelectedCells:**

selectedCol

– (int)**selectedCol**

Returns the column number of the selected Cell, or –1 if no Cells are selected. If Cells in multiple columns are selected, this method returns the number of the last column containing a selected Cell.

See also: – **selectedRow**

selectedRow

– (int)**selectedRow**

Returns the row number of the selected Cell, or –1 if no Cells are selected. If Cells in multiple rows are selected, this method returns the number of the last row containing a selected Cell.

See also: – **selectedCol**

selectText:

– **selectText:***sender*

If *sender* is the next Text object of the Matrix (as set with **setNextText:**), the text in the last selectable text Cell (the one lowest and furthest to the right) is selected; otherwise, the text of the first selectable text Cell is selected. Returns the Cell whose text was selected, the Matrix if such a Cell wasn't found, and **nil** if the Cell was bound but wasn't enabled or wasn't selectable.

See also: – **selectTextAt::**, – **selectText:** (TextField)

selectTextAt::

– **selectTextAt:**(int)*row* **:**(int)*col*

Select the text of the Cell at (*row*, *col*) in the Matrix, if there is such a Cell and its text is selectable. Returns the Cell whose text was selected, the Matrix if such a Cell wasn't found, and **nil** if the Cell was found but wasn't enabled or wasn't selectable.

See also: – **selectText:**, – **selectText:** (TextField)

sendAction

– **sendAction**

If the selected Cell has both an action and a target, its action is sent to its target. If the Cell has an action but no target, its action is sent to the target of the Matrix. If the Cell doesn't have an action, or if there is no selected Cell, the Matrix sends its action to its target.

See also: – **sendDoubleAction**, – **sendAction:to:**, – **action**, – **target**

sendAction:to:

– **sendAction:**(SEL)*theAction* **to:***theTarget*

If both *theAction* and *theAction* are non-null, sends *theAction* to *theTarget*. If *theAction* is null, sends the action of the Matrix to its target. If *theAction* is **nil**, sends *theAction* to the target of the Matrix. Returns **nil** if no target that responds to *theAction* could be found; otherwise returns **self**.

Your code shouldn't normally invoke this method. It is used by event handling methods such as Cell's **trackMouse:inRect:ofView:** to send an action to a target in response to an event within the Matrix.

See also: – **sendAction**, – **sendAction:to:** (Control)

sendAction:to:forAllCells:

– **sendAction:**(SEL)*aSelector*
to:*anObject*
forAllCells:(BOOL)*flag*

Iterates through the Cells in the Matrix, sending *aSelector* to *anObject* for each. *aSelector* must represent a method that takes a single argument: the **id** of the current Cell in the iteration. *aSelector*'s return value must be a BOOL. Iteration begins with the Cell in the upper-left corner of the Matrix, proceeding through all entries in the first row, then on to the next. Returns **self**.

If *aSelector* returns NO for any Cell, this method terminates immediately and return **self**, without sending the message for other Cells. If it returns YES, this method keeps sending the message.

This method is *not* invoked to send action messages to target objects in response to mouse-down events in the Matrix. Instead, you can invoke it if you want to have multiple Cells in a Matrix interact with an object. For example you could use it to verify the titles in a list of items, or to enable a series of radio buttons based on their purpose in relation to *anObject*.

See also: – **sendAction:to:**

sendDoubleAction

– **sendDoubleAction**

If the Matrix has a double-click action, sends that message to the target of the Matrix. If not, then if the selected Cell (as returned by **selectedCell**) has an action, that message is

sent to the selected Cell's target. If the selected Cell also has no action, then the action of the Matrix is sent to the target of the Matrix. This method only sends an action if the selected Cell is enabled. Returns **self**.

Your code shouldn't invoke this method; it's sent in response to a double-click event in the Matrix. You may want to override it to change the search order for an action to send.

See also: – **sendAction**, – **sendAction:to:**, – **ignoreMultiClick:** (Control)

setAction:

– **setAction:**(SEL)*aSelector*

Sets the default action of the Matrix, the message sent for a Cell which has no action of its own. The action of the Matrix is always sent to its target, never to the Cell's target. Returns **self**.

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

setAction:at::

– **setAction:**(SEL)*aSelector*
at:(int)*row*
:(int)*col*

Sets the action of the Cell at (*row*, *col*) to *aSelector*. Returns **self**.

See also: – **setAction:** (ActionCell)

setAutoscroll:

– **setAutoscroll:**(BOOL)*flag*

If *flag* is YES and the Matrix is in a scrolling View, it will be automatically scrolled whenever a the mouse is dragged outside the Matrix after a mouse-down event within its bounds. Returns **self**.

setAutosizeCells:

– **setAutosizeCells:**(BOOL)*flag*

If *flag* is YES, then whenever the Matrix is resized, the sizes of the Cells changes in proportion, keeping the inter-Cell space constant; further, this method verifies that the Cell

sizes and inter-Cell spacing add up to the exact size of the Matrix, adjusting the size of the Cells and updating the Matrix if they don't. If *flag* is NO, then the inter-Cell space changes when the Matrix is resized, with the Cell size remaining constant. Returns **self**.

See also: – **doesAutosizeCells**, – **update** (Control)

setBackgroundColor:

– **setBackgroundColor:**(NXColor)*aColor*

Sets the background color for the Matrix to *aColor*. This color is used to fill the space between Cells or the space behind any non-opaque Cells. Doesn't redraw the Matrix even if autodisplay is on. Returns **self**.

See also: – **backgroundColor**, – **setBackgroundGray:**, – **setCellBackgroundColor:**, – **isAutodisplay** (View)

setBackgroundGray:

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

Sets the background gray level for the Matrix to *value*. This gray level is used to draw the inter-Cell space, or the space behind any non-opaque Cells. If the gray level is –1, the background is transparent (that is, doesn't get drawn). Updates the Matrix if the background gray level changes. Returns **self**.

See also: – **backgroundGray**, – **setBackgroundColor:**, – **setCellBackgroundGray:**, – **update** (Control)

setBackgroundTransparent:

– **setBackgroundTransparent:**(BOOL)*flag*

If *flag* is YES, sets the background gray level of the Matrix to –1 (transparent). If *flag* is NO, set the background gray level to NX_WHITE.

See also: – **isBackgroundTransparent**, – **setBackgroundGray:**

setCellBackgroundColor:

– **setCellBackgroundColor:**(NXColor)*aColor*

Sets the background color for the Cells in the Matrix to *aColor*. This color is used to fill the space behind non-opaque Cells. Doesn't redraw the Matrix even if autodisplay is on. Returns **self**.

See also: – **cellBackgroundColor**, – **setCellBackgroundGray:**, – **setBackgroundColor:**, – **isAutodisplay** (View)

setCellBackgroundGray:

– **setCellBackgroundGray:**(float)*value*

Sets the background gray level for the Cells in the Matrix to *value*. This gray level is used to draw the space behind non-opaque Cells. If the gray level is –1, the Cell background is transparent (that is, doesn't get drawn). Updates the Matrix if the Cell background gray level changes. Returns **self**.

See also: – **cellBackgroundGray**, – **setCellBackgroundColor:**, – **setBackgroundGray:**, – **isAutodisplay** (View)

setCellBackgroundTransparent:

– **setCellBackgroundTransparent:**(BOOL)*flag*

If *flag* is YES, sets the background gray level of Cells in the Matrix to –1 (transparent). If *flag* is NO, set their background gray level to NX_WHITE.

See also: – **isCellBackgroundTransparent**, – **setCellBackgroundGray:**

setCellClass:

– **setCellClass:***classId*

Configures a single Matrix to use instances of *classId* when creating new Cells. *classId* should be the **id** of a subclass of Cell, obtained by sending the **class** message to either the Cell subclass object or to an instance of that subclass. The Cell class is that set with the class method **setCellClass:**; the default Cell class is ActionCell. Returns **self**.

You only need to use this method with Matrices initialized with **initFrame:**, since the other initializers allow you to specify an instance-specific Cell class or Cell prototype.

See also: + **setCellClass:**, – **setPrototype:**, – **initFrame:**

setCellSize:

– **setCellSize:**(const NXSize *)*aSize*

Sets the width and the height of each of the Cells in the Matrix to those in *aSize*. This may change the size of the Matrix. Does not redraw the Matrix, even if autodisplay is on.

See also: – **getCellSize:**, – **calcSize**, – **isAutodisplay** (View)

setDoubleAction:

– **setDoubleAction:**(SEL)*aSelector*

Make *aSelector* the action sent to the target of the Matrix when the user double-clicks a Cell. A double-click action is always sent after the appropriate single-click action; the Cell's if it has one, otherwise the single-click action of the Matrix. Returns **self**.

If a Matrix has no double-click action set, then by default a double-click is treated as a single-click. Setting a double-click action also sets **allowMultiClick**: to YES; be sure to set the Matrix to ignore multiple-clicks if you later remove the double-click action.

See also: – **doubleAction**, – **setAction:**, – **setTarget:**, – **ignoreMultiClick:** (Control)

setEmptySelectionEnabled:

– **setEmptySelectionEnabled:**(BOOL)*flag*

If *flag* is YES, then the Matrix will allow one or zero Cells to be selected. If *flag* is NO, then the Matrix will allow one and only one Cell (not zero Cells) to be selected. This setting has effect only in NX_RADIOMODE.

This method replaces the **allowEmptySel:** method of NeXTSTEP Release 2.

See also: – **isEmptySelectionEnabled**

setEnabled:

– **setEnabled:**(BOOL)*flag*

If *flag* is YES, enables all Cells in the Matrix; if NO, disables all Cells. If autodisplay is on, this redraws the entire Matrix. Returns **self**.

See also: – **isEnabled**, – **setEnabled:** (ActionCell), – **isAutodisplay** (View)

setErrorAction:

– **setErrorAction:**(SEL)*aSelector*

Sets the action sent to the target of the Matrix when the user enters an illegal value in a text Cell for that Cell's entry type (as set by Cell's **setEntryType:** method and checked by Cell's **isEntryAcceptable:** method). Returns **self**.

See also: – **errorAction**, – **setEntryType:** (Cell), – **isEntryAccectable:** (Cell)

setFont:

– **setFont:***fontObject*

Sets the Font for the Matrix to *fontObject*. This will cause all current Cells to have their Font changed to *fontObject* as well as cause all future Cells to have that Font. If autodisplay is on, this redraws the entire Matrix. Returns **self**.

See also: – **font**, – **isAutodisplay** (View)

setIcon:at::

– **setIcon:**(const char *)*iconName*
at:(int)*row*
:(int)*col*

Sets the icon of the Cell at (*row*, *col*) to the NXImage with the name *iconName*. If autodisplay is on, then the Cell is redrawn. Returns **self**.

See also: – **setIcon:** (ButtonCell, Cell), – **isAutodisplay** (View)

setIntercell:

– **setIntercell:**(const NXSize *)*aSize*

Sets the width and the height of the space between Cells to those in *aSize*. Doesn't redraw the Matrix, even if autodisplay is on. Returns **self**.

See also: – **getIntercell:**, – **isAutodisplay** (View)

setMode:

– **setMode:**(int)*aMode*

Sets the selection mode of the Matrix. *aMode* can be one of four constants:

NX_TRACKMODE	Just track the mouse inside the Cells
NX_HIGHLIGHTMODE	Highlight the Cell, then track, then unhighlight
NX_RADIOMODE	Allow no more than one selected Cell
NX_LISTMODE	Allow multiple selected Cells

The behaviors associated with these constants are explained in the class description.

See also: – **mode**

setNextText:

– **setNextText:***anObject*

Sets *anObject* as the object whose text is selected when the user presses Tab while editing the last editable text Cell. *anObject* should respond to the **selectText:** message. If *anObject* also responds to both **selectText:** and **setPreviousText:**, it's sent **setPrevious:** with the receiving Matrix as the argument; this builds a two-way connection, so that pressing Tab in the last text Cell selects *anObject*'s text, and pressing Shift-Tab in *anObject* selects the last text Cell of the Matrix. Returns **self**.

See also: – **setPreviousText:**, – **selectText:**

setPreviousText:

– **setPreviousText:***anObject*

Sets *anObject* as the object whose text is selected when the user presses Shift-Tab while editing the first editable text Cell. *anObject* should respond to the **selectText:** message. Your code shouldn't need to use this method directly, since it's invoked automatically by **setNextText:**. In deference to **setNextText:**, this method doesn't build a two-way connection. Returns **self**.

See also: – **setNextText:**, – **selectText:**

setPrototype:

– **setPrototype:***aCell*

Sets the prototype Cell that is copied whenever a new Cell needs to be made. *aCell* should be an instance of a subclass of Cell. If a Matrix has a prototype Cell, it doesn't use its Cell class object to create new Cells; if you want your Matrix to use its Cell class, invoke this method with **nil** as the argument. The Matrix is considered to own the prototype, and will free it when the Matrix is itself freed; be sure to make a copy of an instance that your code may use elsewhere. Returns the old prototype Cell, or **nil** if there wasn't one.

If you implement your own Cell subclass for use as a prototype with a Matrix, make sure your Cell does the right thing when it receives a **copy** message. For example, Object's **copy** copies only pointers, not what they point to—sometimes this is what it should do, sometimes not. The best way to implement **copy** when you subclass Cell is send **copy** to **super**, then copy instance variable values (for example, title strings) into your subclass instance individually. Also, be careful that freeing the prototype will not damage any of the copies that were made and put into the Matrix (due to shared pointers that are freed, for example).

See also: – **prototype**, – **initFrame:mode:prototype:numRows:numCols:**

setReaction:

– **setReaction:**(BOOL)*flag*

Sent to the Matrix by the target of an action message. If *flag* is NO, prevents the selected Cell from changing back to its previous state; if YES, allows it to revert to its previous state (to reflect unhighlighting, for example). Invoke this from an action method if the action causes the Cell to change in such a way that trying to unhighlight it would be incorrect; for example, if the Cell is deleted or its visual appearance completely changed by the action method. Returns **self**.

setScrollable:

– **setScrollable:**(BOOL)*flag*

Sets all the Cells to be scrollable, so that the text they contain scrolls to remain in view if the user types past the edge of the Cell. Returns **self**.

See also: – **setScrollable:** (Cell)

setSelectionByRect:

– **setSelectionByRect:**(BOOL)*flag*

If *flag* is YES, a rectangle of Cells in the Matrix can be selected by dragging the cursor; if *flag* is NO, such selection isn't possible.

See also: – **isSelectionByRect**, – **setSelectionFrom:to:anchor:lit:**

setSelectionFrom:to:anchor:lit:

– **setSelectionFrom:**(int)*startPos*
to:(int)*endPos*
anchor:(int)*anchorPos*
lit:(BOOL)*flag*

Programmatically selects a range of Cells. *startPos*, *endPos*, and *anchorPos* are Cell positions, counting from 0 at the upper left Cell of the Matrix, rows before columns. For example, the third Cell in the top row would be number 2.

startPos and *endPos* are used to mark where the user would have pressed the mouse button and released it, respectively. *anchorPos* locates the "last selected Cell" with regard to extending the selection by Shift- or Alternate-clicking. Finally, *lit* determines whether Cells selected by this method are highlighted.

See also: – **isSelectionByRect**, – **getSelectedCells:**, – **cellList**

setState:at::

– **setState:**(int)*value*
at:(int)*row*
:(int)*col*

Sets the state of the Cell at row *row* and column *col* to *value*. For radio-mode Matrices, this is identical to **selectCellAt::** except that the state can be set to any arbitrary *value*. If autodisplay is on, redraws the affected Cell; if the Matrix is in radio mode, the Cell is redrawn regardless of the setting of autodisplay. Returns **self**.

See also: – **setState:** (Cell), – **selectCellAt::**, – **isAutodisplay** (View)

setTag:at::

– **setTag:**(int)*anInt*
at:(int)*row*
:(int)*col*

If there's a Cell at (*row*, *col*), sets that Cell's tag to *anInt* and returns **self**.

See also: – **setTag:target:action:at::**, – **setTag:** (ActionCell)

setTag:target:action:at::

– **setTag:**(int)*anInt*
target:*anObject*
action:(SEL)*aSelector*
at:(int)*row*
:(int)*col*

If there's a Cell at (*row*, *col*), sets that Cell's tag, target, and action to *anInt*, *anObject*, and *aSelector*, respectively. Returns **self**.

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

setTarget:

– **setTarget:***anObject*

Sets the target object of the Matrix. This is the object to which actions will be sent for Cells that don't have their own target. Returns **self**.

See also: – **target**, – **action**

setTarget:at::

– **setTarget:***anObject*
at:(int)*row*
:(int)*col*

If there's a Cell at (*row*, *col*), sets that Cell's target to *anObject* and returns **self**.

See also: – **setTag:target:action:at::**, – **setTarget:**, – **setTarget:** (ActionCell)

setTextDelegate:

– **setTextDelegate:***anObject*

Sets the object to which the Matrix will forward messages from the field editor. These messages include **text:isEmpty:**, **textWillEnd:**, **textDidEnd:endChar:**, **textWillChange:**, and **textDidChange:**. Returns **self**.

See also: – **textDelegate**, Text class delegate methods

setTitle:at::

– **setTitle:**(const char *)*aString*
at:(int)*row*
:(int)*col*

If there's a Cell at (*row*, *col*), sets that Cell's title to *aString* and returns **self**.

Note: Use this method only with Matrices that have Cells that respond to **setTitle:**. Not all subclasses of Cell implement this method.

See also: – **setTitle:** (ButtonCell, FormCell)

sizeTo::

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

Resizes the Matrix to *width* and *height*, but doesn't redraw it. If the Matrix has been set to autosize its Cells, each Cell is resized proportionally to the change in size of the Matrix, keeping the inter-Cell spacing constant; if the Matrix doesn't autosize, then the inter-Cell spacing is adjusted, and the Cells remain the same size. If editing is going on in the Matrix, it's aborted; after the Matrix is redrawn, the text is reselected to allow editing to continue. Returns **self**.

See also: – **sizeToCells**, – **sizeToFit**, – **setAutosizeCells:**, – **selectText:**

sizeToCells

– **sizeToCells**

Changes the width and the height of the Matrix frame so that it exactly contains the Cells. Does not redraw the Matrix. Returns **self**.

See also: – **sizeTo::**, – **sizeToFit**

sizeToFit

– **sizeToFit**

Changes the Cell size to accommodate the Cell with the largest contents in the Matrix, then changes the width and the height of the Matrix frame so that it exactly contains the Cells. Doesn't redraw the Matrix. Returns **self**.

See also: – **sizeTo::**, – **sizeToCells**, – **calcCellSize:** (Cell)

target

– **target**

Returns the target of the Matrix. This object receives action messages for Cells that don't have their own target or action, and receives all double-click action messages.

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

textDelegate

– **textDelegate**

Returns the object that receives messages passed on by the Matrix from the field editor. The field editor, as mentioned in the TextField class specification, is the Text object used to draw text in all Cells in a Window.

See also: – **setTextDelegate:**

textDidChange:

– **textDidChange:***textObject*

Passes this message on, with the same argument, to the Text delegate of the Matrix. Override this method if you want your subclass of Matrix to act as the field editor's delegate. Returns **self**.

See also: – **textDidChange:** (Text class delegate method)

textDidEnd:endChar:

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

Invoked by *textObject* (the field editor) when text editing ends. Passes this message on, with the same argument, to the Text delegate of the Matrix, then ends editing for the field editor and checks *whyEnd* to see if an action key (Return or Tab) was pressed. If Return was pressed, an action message is sent as explained in the **sendAction** method description. If Tab was pressed, the next editable text Cell is selected, or if the current Cell is the last one, **selectText:** is sent to the next text if there is one and it responds; failing that, the first selectable text field in the Matrix is selected (that is, the selection cycles within a Matrix with no next text). If Shift-Tab was pressed, a similar sequence is performed in reverse. Returns the object sent the **selectText:** message.

You may want to override this method to interpret more characters (such as the Enter or Escape keys) in ending editing.

See also: – **sendAction**, – **setNextText:**, – **setPreviousText:**, – **textDidEnd:endChar:** (Text class delegate method)

textDidGetKeys:isEmpty:

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

Passes this message on, with the same argument, to the Text delegate of the Matrix. Override this method if you want your subclass of Matrix to act as the field editor's delegate. Returns **self**.

See also: – **textDidGetKeys:isEmpty:** (Text class delegate method)

textWillChange:

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

Invoked automatically during editing to determine if it is OK to edit the selected text. This method checks whether the Cell is editable and sends **textWillChange:** to the TextField's Text delegate to allow it to respond. Returns YES if the text isn't editable; NO if the text is editable but the Text delegate doesn't respond to **textWillChange:**; the text delegate's return value for **textWillChange:** if the Text delegate responds to it.

See also: – **setEditable:** (Cell), – **setTextDelegate:**, – **textWillChange:** (Text class delegate method)

textWillEnd:

– (BOOL)**textWillEnd:***textObject*

Invoked automatically before text editing ends. Checks the text by sending **isEntryAcceptable:** to the Cell being edited. If the entry isn't acceptable, sends the error action to the target. This method is then passed on to the Text delegate with the same argument. The return value is based on whether the entry is acceptable and on the return value from the Text delegate. If the delegate responds to **textWillEnd:**, then the return value is NO only if the entry is acceptable and the delegate returns NO. Otherwise the return value is YES to indicate that editing shouldn't end, and this method generates a beep (to indicate an error in the entry).

See also: – **isEntryAcceptable:** (Cell), – **setTextDelegate:**, – **textWillEnd:** (Text class delegate method)

validateSize:

– **validateSize:**(BOOL)*flag*

If *flag* is YES, then the size information in the Matrix is assumed correct. If *flag* is NO, then **calcSize** will be invoked before any further drawing is done. Returns **self**.

See also: – **calcSize**

write:

– **write:**(NXTypedStream *)*stream*

Writes the receiving Matrix to the typed stream *stream*. Returns **self**.

See also: – **read:**

Menu

Inherits From: Panel : Window : Responder : Object

Declared In: appkit/Menu.h

Class Description

A Menu is a Panel containing a column of MenuCells in a Matrix. Each MenuCell can be configured to send its action message to a target, or to bring up a submenu. When the user clicks a submenu item, the submenu is displayed on the screen, attached to its supermenu so that if the user drags the supermenu, the submenu follows it. A submenu may also be torn away from its supermenu, in which case it displays a close Button.

Exactly one Menu created by the application is designated as the main Menu for the application (with Applications **setMainMenu:** method). This Menu is displayed on top of all other windows whenever the application is active, and should never display a close Button (because it shouldn't have a supermenu).

Many standard Menus are available in Interface Builder, with Menu items that are initialized to work correctly without any additional effort on your part (the Edit, Windows, and Services menus, for example). You can easily use Interface Builder to create other Menu items that display the commands and perform the actions needed by your application.

See the MenuCell and Matrix class specificiations for more details.

Instance Variables

id **supermenu**;
id **matrix**;
id **attachedMenu**;
NXPoint **lastLocation**;
id **reserved**;

```
struct _menuFlags {
        unsigned int sizeFitted:1;
        unsigned int autoupdate:1;
        unsigned int attached:1;
        unsigned int tornOff:1;
        unsigned int wasAttached:1;
        unsigned int wasTornOff:1;
} menuFlags;
```

supermenu	The Menu that this Menu is a submenu of.
matrix	The Matrix that contains the MenuCells.
attachedMenu	The submenu currently attached to this Menu.
lastLocation	Reserved for use by NeXT.
reserved	Reserved for use by NeXT.
menuFlags.sizeFitted	True if the Menu has been sized to fit the Matrix.
menuFlags.autoupdate	True if the Menu accepts automatic update messages.
menuFlags.attached	True if the Menu is attached to its supermenu.
menuFlags.tornOff	True if the Menu has been torn off of its supermenu.
menuFlags.wasAttached	True if the Menu is normally attached to its supermenu.
menuFlags.wasTornOff	True if the Menu was torn off before tracking a popped-up Menu.

Method Types

Creating a Menu zone	+ setMenuZone: + menuZone
Initializing a new Menu	– init – initTitle:
Setting up the Menu commands	– addItem:action:keyEquivalent: – setItemList: – itemList
Finding Menu items	– findCellWithTag:
Building submenus	– setSubmenu:forItem: – submenuAction:

Managing Menu windows	– moveTopLeftTo:: – windowMoved: – getLocation:forSubmenu: – sizeToFit – close
Displaying the Menu	– display – setAutoupdate: – update
Handling events	– mouseDown: – rightMouseDown:
Archiving	– read: – write: – awake

Class Methods

menuZone

+ (NXZone *)**menuZone**

Returns the zone from which new Menus should be allocated. If there isn't one, creates and returns a zone named "Menus." After invoking this method, you should allocate all new Menus from this zone.

See also: – **alloc** (Object), + **setMenuZone:**

setMenuZone:

+ **setMenuZone:**(NXZone *)*zone*

Sets to *zone* the zone from which new Menus should be allocated.

See also: – **alloc** (Object), + **menuZone:**

Instance Methods

addItem:action:keyEquivalent:

– **addItem:**(const char *)*aString*
action:(SEL)*aSelector*
keyEquivalent:(unsigned short)*charCode*

Adds a new command named *aString* to the bottom of the receiving Menu and returns the MenuCell created. The MenuCell's action method is set to *aSelector*, but its target is **nil**. *charCode* is set as the MenuCell's key equivalent. The command name and key equivalent aren't checked for duplications within the same Menu (or any other Menu); be sure to assign them uniquely. The new MenuCell is enabled, but has no tag or alternate title; your code may set these, but should never set a MenuCell's icon.

This method doesn't automatically redisplay the Menu. Upon the next **display** message, the Menu is automatically sized to fit and displayed.

See also: – **setSubmenu:forItem:**

awake

– **awake**

Checks whether an unarchived Menu should have a close Button. Your code shouldn't invoke this method; it's invoked by the **read:** method. Returns **self**.

See also: – **read:**

close

– **close**

Overrides Window's **close** method. Ensures that attached submenus are closed along with the receiver.

See also: – **close** (Window)

display

– **display**

Overrides Window's **display** method so that the Menu is automatically sized to fit its Matrix of items if needed. All Menu methods that change the appearance of the Matrix delay resizing and displaying the Menu until it receives this message.

See also: – **sizeToFit**

findCellWithTag:

– **findCellWithTag:**(int)*aTag*

Returns the MenuCell that has *aTag* as its tag, or **nil** if no such Cell can be found. If your application uses MenuCell tages, each MenuCell should have a unique tag.

See also: – **findCellWithTag:** (Matrix), – **setTag:** (ActionCell)

getLocation:forSubmenu:

– **getLocation:**(NXPoint *)*theLocation* **forSubmenu:***aSubmenu*

Returns the location in screen coordinates at which the lower-left corner of the receiving Menu's submenu should be drawn. Menu invokes this method whenever it brings up a submenu. By default, the submenu is to the right of its supermenu, with its title bar aligned with the supermenu's. Your code need never directly use this method, but may override it to cause the submenu to be attached at a different location.

See also: – **submenuAction:**

init

– **init**

Initializes and returns the receiver, a new instance of Menu, displaying the title "Menu". All other features are as described in the **initTitle:** method below.

See also: – **initTitle:**

initTitle:

– **initTitle:**(const char *)*aTitle*

Initializes and returns the receiver, a new instance of Menu, displaying the title *aTitle*. The Menu is positioned in the upper left corner of the screen, and has no command items. A new Menu must receive an **orderFront:** message to be displayed on the screen; the Application object takes care of this for standard Menus.

The Menu is created as a buffered window, of style NX_MENUSTYLE and button mask NX_CLOSEBUTTON (though a Menu hides its close button until it's torn off from its supermenu). All Menus have an event mask that excludes keyboard events, so they never become the key window or main window.

See also: – **addItem:action:keyEquivalent:**, – **init**

itemList

– **itemList**

Returns the Matrix of MenuCells used by the Menu, which your code can use to add or rearrange command items directly. Be sure to send **sizeToFit** after altering the Matrix, as the Menu won't know that the Matrix has been altered.

See also: – **setItemList:**, – **sizeToFit**

mouseDown:

– **mouseDown:**(NXEvent *)*theEvent*

Overrides the Responder method to catch a mouse-down event instead of passing it along, so that the Menu can track the mouse itself and manage display of its submenus properly. MenuCell's **trackMouse:inRect:ofView:** sends this message. Returns **self**.

See also: – **rightMouseDown:**, – **trackMouse:inRect:ofView:** (MenuCell)

moveTopLeftTo::

– **moveTopLeftTo:**(NXCoord)*x* **:**(NXCoord)*y*

Moves the top left corner of the Menu to the position on the screen defined (in screen coordinates) by *x* and *y*. This method is overriden fromWindow's to resize the Matrix if needed before redisplaying the Menu at the new location.

See also: – **moveTo::** (Window)

read:

– **read:**(NXTypedStream *)*stream*

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

See also: – **awake**, – **write:**

rightMouseDown:

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

Pops the menu up under the cursor position in *theEvent*. Before doing so, this method saves the current state of the Menu (including selected cells, attached submenus, menu positions, and so on). The menu is popped up with no Cells selected or submenus attached. The Menu is tracked as for a **mouseDown:** event. On mouse-up, the Menu's state is restored so that the original Menu arrangment on screen isn't changed. Returns self.

See also: – **mouseDown:**

setAutoupdate:

– **setAutoupdate:**(BOOL)*flag*

If *flag* is YES, the Menu will invoke the update action for each MenuCell whenever it receives an **update** message—usually sent by the Application object when autoupdating of windows is enabled. If NO, the Menu doesn't update its MenuCells on receiving an **update** message.

See also: – **update**, – **setAutoupdate:** (Application), – **setUpdateAction:** (MenuCell)

setItemList:

– **setItemList:***aMatrix*

Sets the Menu's Matrix of items to *aMatrix*. A following **display** message will size the Menu to fit the new Matrix before drawing. Returns the old Matrix.

See also: – **itemList**, – **display**

setSubmenu:forItem:

– **setSubmenu:***aMenu* **forItem:***aCell*

Sets *aMenu* as the submenu of the receiver, controlled by the MenuCell *aCell*. *aCell*'s target is set to *aMenu*, its action to **submenuAction:** and its icon to the arrow indicating that it brings up a submenu. Doesn't remove *aCell*'s key equivalent. If *aMenu* was on screen, it won't be removed from the screen or moved until it's first brought up as a submenu. Returns *aCell*.

See also: – **submenuAction:**

sizeToFit

– **sizeToFit**

Sizes the Menu's Matrix to its MenuCells, so that all items fit in as small a rectangle as possible, and then fits the Menu to the resized Matrix. Use this method after you've added or altered items by sending messages directly to the Matrix. When the Menu is resized, its upper left corner remains fixed. After performing any necessary resizing, this method redisplays the Menu.

See also: – **sizeToFit** (Matrix), – **display**

submenuAction:

– **submenuAction:***sender*

Action method sent to a submenu associated with an entry in a Menu. If sender's Window is a visible Menu, the receiver attaches and displays itself as a submenu of the sender's Menu; otherwise, does nothing. *sender* should be the Matrix containing the MenuCell that brings up the submenu. Returns **self**.

See also: – **setSubmenu:forItem:**

update

– **update**

Updates the Menu's items. If the Menu has been set to autoupdate, this method gets the update action method for each of its MenuCells and sends that method to the first of the following that responds to it: the Menu's delegate, NXApp, or NXApp's delegate. If a MenuCell's update action returns YES, that MenuCell is redrawn.

See also: – **setAutoupdate:** – **setUpdateAction:** (MenuCell)

windowMoved:

– **windowMoved:**(NXEvent *)*theEvent*

Overrides the Window method to implement tear-off Menu behavior. When a submenu is torn off, the item selected in its supermenu is unhighlighted. The submenu is flagged as detached, is moved to the appropriate window level, and displays its close Button. Returns **self**.

See also: – **windowMoved:** (Window)

write:

– **write:**(NXTypedStream *)*stream*

Writes the receiving Menu to the typed stream *stream*. Returns **self**.

See also: – **read:**

MenuCell

Inherits From: ButtonCell : ActionCell : Cell : Object

Declared In: appkit/MenuCell.h

Class Description

MenuCell is a subclass of ButtonCell that appears in Menus. MenuCells draw their text left-justified and show an optional key equivalent or submenu arrow on the right. See the Menu class specification for more information. PopUpList is a subclass of Menu that uses MenuCells for its entries.

Instance Variables

SEL **updateAction**;

updateAction	Method to make the MenuCell reflect its applicability to the application's state

Method Types

Initializing a new MenuCell	– init – initTextCell:
Setting the update action	– setUpdateAction:forMenu: – updateAction
Checking for a submenu	– hasSubmenu
Tracking the mouse	– trackMouse:inRect:ofView:
Setting user key equivalents	+ useUserKeyEquivalents: – userKeyEquivalent
Archiving	– read: – write:

Class Methods

useUserKeyEquivalents:

+ **useUserKeyEquivalents:**(BOOL)*flag*

Sets whether MenuCells conform to user preferences for key equivalents. If *flag* is YES, then MenuCells replace their key equivalents with those in the user's defaults. If NO, the key equivalents originally assigned to the MenuCells are used.

See also: – **userKeyEquivalent**

Instance Methods

hasSubmenu

– (BOOL)**hasSubmenu**

Return YES if the MenuCell brings up a submenu, NO otherwise.

See also: – **setSubmenu:forItem:** (Menu)

init

– **init**

Initializes and returns the receiver, a new instance of MenuCell, with the default title "MenuItem".

See also: – **initTextCell:**

initTextCell:

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

Initializes and returns the receiver, a new instance of MenuCell, with *aString* as its title. This is the designated initializer for the MenuCell class; override this method if you create a subclass of MenuCell that performs its own initialization. You should never use Cell's **initIconCell:** with MenuCells.

See also: – **init**

read:

– **read:**(NXTypedStream *)*stream*

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

See also: – **write:**

setUpdateAction:forMenu:

– **setUpdateAction:**(SEL)*aSelector* **forMenu:***aMenu*

Sets the MenuCell's update action to *aSelector*, and sets *aMenu* to autoupdate. A MenuCell's update action should be a method that takes one **id**, the MenuCell to be updated, as the argument, and returns YES if the MenuCell needs to be redrawn, NO if it doesn't. The update action should alter the MenuCell if needed to reflect its applicability in the current state of the application. This may involve enabling or disable the MenuCell, changing its title, or setting its state. The MenuCell's Menu sends the update action to the first of the following that responds to it: the Menu's delegate, the Application object, or the Application object's delegate. Returns **self**.

See also: – **update** (Menu), – **setAutoupdate:** (Menu),
– **updateWindows:** (Application)

trackMouse:inRect:ofView:

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

Passes *theEvent* as the argument of a **mouseDown:** message to the receiver's Menu. Menus handle all mouse tracking themselves.

See also: – **mouseDown:** (Menu)

updateAction

– (SEL)**updateAction**

Returns the update action used to update the receiver's state in response to an automatic application update.

See also: – **setUpdateAction:**

userKeyEquivalent

– (unsigned short)**userKeyEquivalent**

If the MenuCell class has been configured to use user key equivalents, returns the user-assigned key equivalent for the receiving MenuCell.

See also: + **useUserKeyEquivalents:**

write:

– **write:**(NXTypedStream *)*stream*

Writes the receiving MenuCell to the typed stream *stream*. Returns **self**.

See also: – **read:**

NXBitmapImageRep

Inherits From: NXImageRep : Object

Declared In: appkit/NXBitmapImageRep.h

Class Description

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

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

Setting Up an NXBitmapImageRep

A new NXBitmapImageRep is passed bitmap data for an image—or told where to find it—when it's first initialized:

- TIFF data can be read from a stream, from a file, or from a section of the __TIFF segment of the application executable. If it's stored in a section or a separate file, the object will delay reading the data until it's needed.
- Raw bitmap data is placed in buffers, and pointers to the buffers are passed to the object.

An NXBitmapImageRep can also be created from bitmap data that's read from an existing (already rendered) image. The object created from this data is able to reproduce the image.

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

TIFF Compression

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

LZW	Compresses and decompresses without information loss, achieving compression ratios of anywhere from 2:1 to 3:1. It may be somewhat slower to compress and decompress than the PackBits scheme.
PackBits	Compresses and decompresses without information loss, but may not achieve the same compression ratios as LZW.
JPEG	Compresses and decompresses with some information loss, but can achieve compression ratios anywhere from 10:1 to 100:1. The ratio is determined by a user-settable factor ranging from 1.0 to 255.0, with higher factors yielding greater compression. More information is lost with greater compression, but 15:1 compression is safe for publication quality. Some images can be compressed even more. JPEG compression can be used only for images that specify at least 4 bits per sample.
CCITTFAX	Compresses and decompresses 1 bit grayscale images using international fax compression standards CCITT3 and CCITT4.

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

Instance Variables

None declared in this class.

Method Types

Initializing a new NXBitmapImageRep object

- initFromSection:
- initFromFile:
- initFromStream:
- initData:fromRect:
- initData:pixelsWide:pixelsHigh:bitsPerSample: samplesPerPixel:hasAlpha:isPlanar:colorSpace: bytesPerRow:bitsPerPixel:
- initDataPlanes:pixelsWide:pixelsHigh: bitsPerSample:samplesPerPixel:hasAlpha: isPlanar:colorSpace:bytesPerRow:bitsPerPixel:

Creating a List of NXBitmapImageReps

+ newListFromSection:
+ newListFromSection:zone:
+ newListFromFile:
+ newListFromFile:zone:
+ newListFromStream:
+ newListFromStream:zone:

Reading information from a rendered image

+ sizeImage:
+ sizeImage:pixelsWide:pixelsHigh:bitsPerSample:
  samplesPerPixel:hasAlpha:isPlanar:colorSpace:

Copying and freeing an NXBitmapImageRep

– copyFromZone:
– free

Getting information about the image

– bitsPerPixel
– samplesPerPixel
– bitsPerSample (NXImageRep)
– isPlanar
– numPlanes
– numColors (NXImageRep)
– hasAlpha (NXImageRep)
– bytesPerPlane
– bytesPerRow
– colorSpace
– pixelsWide (NXImageRep)
– pixelsHigh (NXImageRep)

Getting image data

– data
– getDataPlanes:

Drawing the image

– draw
– drawIn:
– drawAt: (NXImageRep)

Producing a TIFF representation of the image

– writeTIFF:
– writeTIFF:usingCompression:
– writeTIFF:usingCompression:andFactor:

Setting/checking compression types

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

Checking unpacked data handling	+ setUnpackedImageDataAcceptable: + isUnpackedImageDataAcceptable
Archiving	– read: – write:

Class Methods

getTIFFCompressionTypes:count:

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

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

The following compression types are supported:

Constant	Value	Usage
NX_TIFF_COMPRESSION_NONE	1	
NX_TIFF_COMPRESSION_CCITTFAX3	3	1 bps images only
NX_TIFF_COMPRESSION_CCITTFAX4	4	1 bps images only
NX_TIFF_COMPRESSION_LZW	5	
NX_TIFF_COMPRESSION_JPEG	6	
NX_TIFF_COMPRESSION_NEXT	32766	Input only
NX_TIFF_COMPRESSION_PACKBITS	32773	
NX_TIFF_COMPRESSION_OLDJPEG	32865	Input only

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

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

isUnpackedImageDataAcceptable

+ (BOOL)**isUnpackedImageDataAcceptable**

Returns YES if the NXBitmapImageRep class can accept unpacked image data. You can set the value returned by this method through the **setUnpackedImageDataAcceptable:** class method.

See also: + **setUnpackedImageDataAcceptable:**

localizedNameForTIFFCompressionType:

+ (const char *)**localizedNameForTIFFCompressionType:**(int)*compression*

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

The returned string belongs to the NXBitmapImageRep class; don't attempt to alter or free it.

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

newListFromFile:

+ (List *)**newListFromFile:**(const char *)*filename*

Creates one new NXBitmapImageRep instance for each TIFF image specified in the *filename* file, and returns a List object containing all the objects created. If no NXBitmapImageReps can be created (for example, if *filename* doesn't exist or doesn't contain TIFF data), **nil** is returned. The List should be freed when it's no longer needed.

Each new NXBitmapImageRep is initialized by the **initFromFile:** method, which reads information about the image from *filename*, but not the image data. The data will be read when it's needed to render the image.

See also: + **newListFromFile:zone:**, – **initFromFile:**

newListFromFile:zone:

+ (List *)**newListFromFile:**(const char *)*filename* **zone:**(NXZone *)*aZone*

Returns a List of new NXBitmapImageRep instances, just as **newListFromFile:** does, except that the List object and the NXBitmapImageReps are allocated from memory located in *aZone*.

See also: + **newListFromFile:**, – **initFromFile:**

newListFromSection:

+ (List *)**newListFromSection:**(const char *)*name*

Creates one new NXBitmapImageRep instance for each TIFF image specified in the *name* section of the __TIFF segment in the executable file or in the *name* file in the application bundle, and returns a List object containing all the objects created. If not even one NXBitmapImageRep can be created (for example, if the *name* section doesn't exist or doesn't contain TIFF data), **nil** is returned. The List should be freed when it's no longer needed.

Each new NXBitmapImageRep is initialized by the **initFromSection:** method, which reads information about the image from the section, but doesn't read image data. The data will be read when it's needed to render the image.

See also: + **newListFromSection:zone:**, – **initFromSection:**

newListFromSection:zone:

+ (List *)**newListFromSection:**(const char *)*name* **zone:**(NXZone *)*aZone*

Returns a List of new NXBitmapImageRep instances, just as **newListFromSection:** does, except that the List object and the NXBitmapImageReps are allocated from memory located in *aZone*.

See also: + **newListFromSection:**, – **initFromSection:**

newListFromStream:

+ (List *)**newListFromStream:**(NXStream *)*stream*

Creates one new NXBitmapImageRep instance for each TIFF image that can be read from *stream*, and returns a List object containing all the objects created. If not even one NXBitmapImageRep can be created (for example, if the *stream* doesn't contain TIFF data), **nil** is returned. The List should be freed when it's no longer needed.

The data is read and each new object initialized by the **initFromStream:** method.

See also: + **newListFromStream:zone:**, – **initFromStream:**

newListFromStream:zone:

+ (List *)**newListFromStream:**(NXStream *)*stream* **zone:**(NXZone *)*aZone*

Returns a List of new NXBitmapImageRep instances, just as **newListFromStream:** does, except that the NXBitmapImageReps and the List object are allocated from memory located in *aZone*.

See also: + **newListFromStream:**, – **initFromStream:**

setUnpackedImageDataAcceptable:

+ (void)**setUnpackedImageDataAcceptable:**(BOOL)*flag*

If *flag* is YES, sets the receiver to accept unpacked image data.

See also: + **isUnpackedImageDataAcceptable**

sizeImage:

+ (int)**sizeImage:**(const NXRect *)*rect*

Returns the number of bytes that would be required to hold bitmap data for the rendered image bounded by the *rect* rectangle. The rectangle is located in the current window and is specified in the current coordinate system.

See also: + **sizeImage:pixelsWide:pixelsHigh:bitsPerSample:samplesPerPixel: hasAlpha:isPlanar:colorSpace:**, – **initData:fromRect:**

sizeImage:pixelsWide:pixelsHigh:bitsPerSample:samplesPerPixel: hasAlpha:isPlanar:colorSpace:

+ (int)**sizeImage:**(const NXRect *)*rect*
  pixelsWide:(int *)*width*
  pixelsHigh:(int *)*height*
  bitsPerSample:(int *)*bps*
  samplesPerPixel:(int *)*spp*
  hasAlpha:(BOOL *)*alpha*
  isPlanar:(BOOL *)*config*
  colorSpace:(NXColorSpace *)*space*

Returns the number of bytes that would be required to hold bitmap data for the rendered image bounded by the *rect* rectangle. The rectangle is located in the current window and is specified in the current coordinate system.

Every argument but *rect* is a pointer to a variable where the method will write information about the image. For an explanation of the information provided, see the description of the **initDataPlanes:...** method

See also: – **initDataPlanes:pixelsWide:pixelsHigh:bitsPerSample: samplesPerPixel:hasAlpha:isPlanar:colorSpace:bytesPerRow:bitsPerPixel:**

Instance Methods

bitsPerPixel

– (int)**bitsPerPixel**

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

Warning: Currently, an NXBitmapImageRep cannot render an image that has empty memory separating pixel specifications.

bytesPerPlane

– (int)**bytesPerPlane**

Returns the number of bytes in each plane or channel of data. This will be figured from the number of bytes per row and the height of the image.

See also: – **bytesPerRow**

bytesPerRow

– (int)**bytesPerRow**

Returns the minimum number of bytes required to specify a scan line (a single row of pixels spanning the width of the image) in each data plane. If not explicitly set to another value (in the **initData:...** or **initDataPlanes:...** method), this will be figured from the width of the

image, the number of bits per sample, and, if the data is in a meshed configuration, the number of samples per pixel. It can be set to another value to indicate that each row of data is aligned on word or other boundaries.

Warning: Currently, an NXBitmapImageRep can't render an image that has empty space at the end of a scan line.

canBeCompressedUsing:

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

Tests whether the receiver can be compressed by *compression* type. Compression types are defined in **appkit/tiff.h**. This method returns YES if the receiver's data matches *compression*; for example, if *compression* is NX_TIFF_COMPRESSION_CCITTFAX3, then the data must be one bit-per-sample and one sample-per-pixel. It returns NO if the data doesn't match *compression* or if *compression* is unsupported.

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

colorSpace

– (NXColorSpace)**colorSpace**

Returns one of the following enumerated values, which indicate how bitmap data is to be interpreted:

NX_OneIsBlackColorSpace	A gray scale between 1 (black) and 0 (white)
NX_OneIsWhiteColorSpace	A gray scale between 0 (black) and 1 (white)
NX_RGBColorSpace	Red, green, and blue color values
NX_CMYKColorSpace	Cyan, magenta, yellow, and black color values

See also: – **numColors** (NXImageRep)

copyFromZone:

– **copyFromZone:**(NXZone *)*zone*

Returns a new NXBitmapImageRep instance that's an exact copy of the receiver. The new object is allocated from *zone*. It will have its own copy of the bitmap data, also allocated from *zone*, unless the receiver merely references the data. In that case, both objects will reference the same data.

See also: – **copy** (Object)

data

– (unsigned char *)**data**

Returns a pointer to the bitmap data. If the data is in planar configuration, this pointer will be to the first plane. To get separate pointers to each plane, use the **getDataPlanes:** method.

See also: – **getDataPlanes:**

draw

– (BOOL)**draw**

Renders the image at (0.0, 0.0) in the current coordinate system on the current device using the appropriate PostScript imaging operator. This method returns YES if successful in producing the image, and NO if not.

See also: – **drawAt:** (NXImageRep), – **drawIn:**

drawIn:

– (BOOL)**drawIn:**(const NXRect *)*rect*

Renders the image so that it fits inside the rectangle referred to by *rect*. The current coordinate system is translated and scaled so the image will appear at the right location and fit within the rectangle. The **draw** method is then invoked to render the image. This method passes through the return value of the **draw** method, which indicates whether the image was successfully drawn.

The coordinate system is not restored after it has been altered.

See also: – **draw**, – **drawAt:** (NXImageRep)

free

– **free**

Deallocates the NXBitmapImageRep. This method will not free any bitmap data that the object merely references—that is, raw data that was passed to it in a **initData:...** or **initDataPlanes:...** message.

getCompression:andFactor:

– (void)**getCompression:**(int *)*compression* **andFactor:**(float *)*factor*

Returns by reference the receiver's compression type and compression factor. Use this method to get information on the compression type for the source image data. *compression* represents the compression type used on the data, and corresponds to one of the values returned by the class method **getTIFFCompressionTypes:count:**. *factor* is usually a value between 0.0 and 255.0, with 0.0 representing no compression.

See also: + **getTIFFCompressionTypes:count:**, – **setCompression:andFactor:**

getDataPlanes:

– **getDataPlanes:**(unsigned char **)*thePlanes*

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

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

See also: – **data**, – **isPlanar**

init

Generates an error message. This method cannot be used to initialize an NXBitmapImageRep. Instead, use one of the **initFrom...** or **initData...** methods.

See also: – **initFromSection:**, – **initFromFile:**, – **initFromStream:**, – **initDataPlanes:...**, – **initData:...**

initData:fromRect:

– **initData:**(unsigned char *)*data* **fromRect:**(const NXRect *)*rect*

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

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

If *data* is NULL, the NXBitmapImageRep will allocate enough memory to hold bitmap data for the image. In this case, the buffer will belong to the object and will be freed when the object is freed.

If *data* is not NULL, you must make sure the buffer is large enough to hold the image bitmap. You can determine how large it needs to be by sending a **sizeImage:** message for the same rectangle. The NXBitmapImageRep will only reference the data in the buffer; the buffer won't be freed when the object is freed.

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

See also: + **sizeImage:**

initData:pixelsWide:pixelsHigh:bitsPerSample:samplesPerPixel: hasAlpha:isPlanar:colorSpace:bytesPerRow:bitsPerPixel:

– **initData:**(unsigned char *)*data*
pixelsWide:(int)*width*
pixelsHigh:(int)*height*
bitsPerSample:(int)*bps*
samplesPerPixel:(int)*spp*
hasAlpha:(BOOL)*alpha*
isPlanar:(BOOL)*config*
colorSpace:(NXColorSpace)*space*
bytesPerRow:(int)*rowBytes*
bitsPerPixel:(int)*pixelBits*

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

data points to a buffer containing raw bitmap data. If the data is in planar configuration (*config* is YES), all the planes must follow each other in the same buffer. The **initDataPlanes:...** method can be used instead of this one if there are separate buffers for each plane.

If *data* is NULL, this method allocates a data buffer large enough to hold the image described by the other arguments. You can then obtain a pointer to this buffer (with the

data or **getDataPlanes:** method) and fill in the image data. In this case the buffer will belong to the object and will be freed when it's freed.

If *data* is not NULL, the object will only reference the image data; it won't copy it. The buffer won't be freed when the object is freed.

All the other arguments to this method are the same as those to **initDataPlanes:...**. See that method for descriptions.

See also: – initDataPlanes:pixelsWide:pixelsHigh:bitsPerSample: samplesPerPixel:hasAlpha:isPlanar:colorSpace:bytesPerRow:bitsPerPixel:

initDataPlanes:pixelsWide:pixelsHigh:bitsPerSample:samplesPerPixel: hasAlpha:isPlanar:colorSpace:bytesPerRow:bitsPerPixel:

– **initDataPlanes:**(unsigned char **)*planes*
pixelsWide:(int)*width*
pixelsHigh:(int)*height*
bitsPerSample:(int)*bps*
samplesPerPixel:(int)*spp*
hasAlpha:(BOOL)*alpha*
isPlanar:(BOOL)*config*
colorSpace:(NXColorSpace)*space*
bytesPerRow:(int)*rowBytes*
bitsPerPixel:(int)*pixelBits*

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

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

If the data is in meshed configuration (*config* is NO), only the first buffer is read. The **initData:...** method can be used instead of this one for data in meshed configuration.

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

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

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

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

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

NX_OneIsBlackColorSpace	A gray scale between 1 (black) and 0 (white)
NX_OneIsWhiteColorSpace	A gray scale between 0 (black) and 1 (white)
NX_RGBColorSpace	Red, green, and blue color values
NX_CMYKColorSpace	Cyan, magenta, yellow, and black color values

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

  Normally, *rowBytes* can be figured from the *width* of the image, the number of bits per pixel in each sample (*bps*), and, if the data is in a meshed configuration, the number of samples per pixel (*spp*). However, if the data for each row is aligned on word or other boundaries, it may have been necessary to allocate more memory for each row than there

is data to fill it. *rowBytes* lets the object know whether that's the case. If *rowBytes* is 0, the NXBitmapImageRep assumes that there's no empty space at the end of a row.

Warning: Currently, an NXBitmapImageRep cannot render an image that has empty space at the end of a scan line.

- *pixelBits* informs the NXBitmapImageRep how many bits are actually allocated per pixel in each plane of data. If the data is in planar configuration, this normally equals *bps* (bits per sample). If the data is in meshed configuration, it normally equals *bps* times *spp* (samples per pixel). However, it's possible for a pixel specification to be followed by some meaningless bits (empty space), as may happen, for example, if pixel data is aligned on byte boundaries. Currently, an NXBitmapImageRep cannot render an image if this is the case.

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

This method is the designated initializer for NXBitmapImageReps that handle raw image data.

See also: – **initData:pixelsWide:pixelsHigh:...**

initFromFile:

– **initFromFile:**(const char *)*filename*

Initializes the receiver, a newly allocated NXBitmapImageRep object, with the TIFF image found in the *filename* file. This method reads some information about the image from *filename*, but not the image itself. Image data will be read when it's needed to render the image.

If the new object can't be initialized for any reason (for example, *filename* doesn't exist or doesn't contain TIFF data), this method frees it and returns **nil**. Otherwise, it returns **self**.

This method is the designated initializer for NXBitmapImageReps that read image data from a file.

See also: + **newListFromFile:**, – **initFromSection:**

initFromSection:

– **initFromSection:**(const char *)*name*

Initializes the receiver, a newly allocated NXBitmapImageRep object, with the TIFF image found in the *name* section in the __TIFF segment of the application executable or the *name* file in the application bundle. This method reads some information about the image from

the section, but not the image itself. Image data is read only when it's needed to render the image.

If the new object can't be initialized for any reason (for example, the *name* section doesn't exist or doesn't contain TIFF data), this method frees it and returns **nil**. Otherwise, it returns **self**.

This method is the designated initializer for NXBitmapImageReps that read image data from the application's executable file or bundle.

See also: + **newListFromSection:**, – **initFromFile:**

initFromStream:

– **initFromStream:**(NXStream *)*stream*

Initializes the receiver, a newly allocated NXBitmapImageRep object, with the TIFF image read from *stream*. If the new object can't be initialized for any reason (for example, *stream* doesn't contain TIFF data), this method frees it and returns **nil**. Otherwise, it returns **self**.

This method is the designated initializer for NXBitmapImageReps that read image data from a stream.

See also: + **newListFromStream:**

isPlanar

– (BOOL)**isPlanar**

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

See also: – **samplesPerPixel**

numPlanes

– (int)**numPlanes**

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

See also: – **isPlanar**, – **samplesPerPixel**, – **hasAlpha**, – **numColors** (NXImageRep)

read:

– **read:**(NXTypedStream *)*stream*

Reads the NXBitmapImageRep from the typed stream *stream*.

See also: – **write:**

samplesPerPixel

– (int)**samplesPerPixel**

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

See also: – **hasAlpha**, – **numColors** (NXImageRep)

setCompression:andFactor:

– (void)**setCompression:**(int)*compression* **andFactor:**(float)*factor*

Sets the receiver's compression type and compression factor. *compression* is one of the supported compression types listed in the **getTiffCompressionTypes:count:** class method description. *factor* is a compression factor, usually between 0.0 (no compression) and 255.0 (maximum compression).

When an NXBitmapImageRep is created, the instance stores the compression type and factor for the source data. When the data is subsequently saved, **writeTIFF:** tries to use the stored compression type and factor. Use this method to change the compression type and factor.

See also: + **getTiffCompressionTypes:count:**, – **getCompression:andFactor:**

write:

– **write:**(NXTypedStream *)*stream*

Writes the NXBitmapImageRep to the typed stream *stream*.

See also: – **read:**

writeTIFF:

– **writeTIFF:**(NXStream *)*stream*

Writes a TIFF representation of the image to *stream*. This method invokes **writeTIFF:usingCompression:andFactor:** using the stored compression type and factor retrieved from the initial image data or changed using **setCompression:andFactor:**. If the stored compression type isn't supported for writing TIFF data (for example, NX_TIFF_COMPRESSION_NEXT), the stored compression is changed to NX_TIFF_COMPRESSION_NONE and the compression factor to 0.0 before invoking **writeTIFF:usingCompression:andFactor:**

See also: – **getCompression:andFactor:**, – **setCompression:andFactor:**, **writeTIFF:usingCompression:andFactor:**

writeTIFF:usingCompression:

– **writeTIFF:**(NXStream *)*stream* **usingCompression:**(int)*compression*

Writes a TIFF representation of the image to *stream*, compressing the data according to the *compression* scheme. If *compression* is NX_TIFF_COMPRESSION_JPEG, the default compression factor is used. This and the other *compression* constants are listed under the next method.

See also: – **writeTIFF:usingCompression:andFactor:**

writeTIFF:usingCompression:andFactor:

– **writeTIFF:**(NXStream *)*stream*
usingCompression:(int)*compression*
andFactor:(float)*factor*

Writes a TIFF representation of the image to *stream*. If the stream isn't currently positioned at location 0, this method assumes that it contains another TIFF image. It will try to append the TIFF representation it writes to that image. To do this, it must read the header of the image already in the stream. Therefore, the stream must be opened with NX_READWRITE permission.

The second argument, *compression*, indicates the compression scheme to use. It should be one of the following constants:

NX_TIFF_COMPRESSION_NONE	No compression
NX_TIFF_COMPRESSION_LZW	LZW compression
NX_TIFF_COMPRESSION_PACKBITS	PackBits compression
NX_TIFF_COMPRESSION_JPEG	JPEG compression
NX_TIFF_COMPRESSION_CCITTFAX3	CCITT Level 3 fax (1 bps data only)
NX_TIFF_COMPRESSION_CCITTFAX4	CCITT Level 4 fax (1 bps data only)

The third argument, *factor*, is used in the JPEG scheme to determine the degree of compression. If *factor* is 0.0, the default compression factor of 10.0 will be used. Otherwise, *factor* should fall within the range 1.0–255.0, with higher values yielding greater compression but also greater information loss.

The compression schemes are discussed briefly in the class description, above.

NXBrowser

Inherits From: Control : View : Responder : Object

Declared In: appkit/NXBrowser.h

Class Description

NXBrowser provides a user interface for displaying and selecting items from a list of data, or from hierarchically organized lists of data such as directory paths. When working with a hierarchy of data, the levels are displayed in columns, which are numbered from left to right, beginning with 0. Each column consists of a ScrollView or ClipView containing a Matrix filled with NXBrowserCells. NXBrowser relies on a delegate to provide the data in its NXBrowserCells. See the BrowserCell class description for more on its implementation.

Browser Selection

An entry in an NXBrowser's column can be either a branch node (such as a directory) or a leaf node (such as a file). When the user selects a single branch node entry in a column, the NXBrowser sends itself the **addColumn** message, which messages its delegate to load the next column. The user's selection can be represented as a character string, much like a UNIX file's pathname; however, the separator can be set to any character, not just '/'. An NXBrowser can be set to allow selection of multiple entries in a column, or to limit selection to a single entry. When set for multiple selection, it can also be set to limit multiple selection to leaf nodes only, or to allow selection of both types of nodes together.

As a subclass of Control, NXBrowser has a target object and action message. Each time the user selects one or more entries in a column, the action message is sent to the target. NXBrowser also adds an action to be sent when the user double-clicks on an entry, which allows the user to select items without any action being taken, and then double-click to invoke some useful action such as opening a file.

User Interface Features

The user interface features of an NXBrowser can be changed in a number of ways. Columns in an NXBrowser may have up and down scroll buttons, scroll bars, both, or neither; the NXBrowser itself may or may not have left and right scroll buttons or a scroll bar. You generally shouldn't create an NXBrowser without scrollers; if you do, you must

make sure the bounds rectangle of the NXBrowser is large enough that all its rows and columns can be displayed. An NXBrowser's columns may be bordered and titled, bordered and untitled, or unbordered and untitled. A column's title may be taken from the selected entry in the column to its left, or may be provided explicitly by the NXBrowser or its delegate. Interface Builder provides easier ways to set many of the user interface features described previously.

NXBrowser's Delegate

NXBrowser requires a delegate to provide it with data for display. The delegate is responsible for providing the data and for setting each item as a branch or leaf node, enabled or disabled. It can also receive notification of events like scrolling and requests for validation of columns that may have changed. You can implement one of three delegate types—normal, lazy, or very lazy—depending on your needs for performance and memory use. An NXBrowser can determine what type of delegate it has by which methods it responds to. To implement a delegate, you implement the normal, lazy, or very lazy methods described below. Two methods, **browser:fillMatrix:inColumn:** and **browser:getNumRowsInColumn:**, are mutually exclusive; you can implement one or the other, but not both.

A normal delegate loads an entire column of its NXBrowser at once, with the **browser:fillMatrix:inColumn:** method. A normal delegate is useful for small sets of data, since it can quickly load each set without much delay, and since each set takes up little memory. A normal delegate creates all of the Cells in a column and fills them with appropriate information, including the title for the Cell, whether it's a node or a leaf, and whether it's enabled or not.

Note: Though called "normal," a normal delgate is not really the most commonly useful. As stated, it's primary useful for small, static sets of data. Lazy and very lazy delegates, described below, are much more flexible, and often end up being easier to program, since data only has to be accessed as it's needed by the NXBrowser.

A lazy delegate creates an entire column for an NXBrowser, but only fills the Cells in the column as requested by the NXBrowser. It must implement both the **browser:fillMatrix:inColumn:** and **browser:loadCell:atRow:inColumn:** methods. When filling a column, it only needs to create the Cells, though it may actually fill them in partially or completely, perhaps setting only the title in order to sort the items. A lazy delegate is useful for fairly large sets of data that would take a long time to load completely. For example, a file system would be well served by a lazy delegate; it could fill each column with the names of all files in that directory, but only when the NXBrowser is about to display a particular Cell would the delegate check whether the file for that Cell is actually a directory (to set the Cell as a node or leaf), and whether the user has permission to access that file or directory (to set the Cell as enabled or disabled).

A very lazy delegate is responsible only for informing its NXBrowser how many items are in a particular column and for loading each Cell on request, with the **browser: getNumRowsInColumn:** and **browser:loadCell:atRow:inColumn:** methods. Very lazy delegates make spare use of memory by not loading a Cell for an entry until it's about to be displayed; this is useful for large, potentially open-ended data spaces that are already sorted, or simply don't need to be sorted. A very lazy delegate is also a good candidate for browsing a file system, provided that the file names can be loaded in the proper positions in a browser column based on their ordering.

An NXBrowser's delegate is also useful for manipulating the data in the NXBrowser on the request of another object. For example, a panel may have a browser with some buttons for adding and deleting entries. Instead of having the entire column reloaded when an entry is added or deleted, the delegate can directly access the NXBrower's Matrix for the selected column, adding or removing Cells, and then invoking NXBrowser's **displayColumn:** method to redraw that column only.

Instance Variables

id **target**;
id **delegate**;
SEL **action**;
SEL **doubleAction**;
id **matrixClass**;
id **cellPrototype**;
unsigned short **pathSeparator**;

target	The object notified by NXBrowser when one or more items are selected in a column.
delegate	The object providing the data which is browsed by the NXBrowser.
action	The message sent to the target when one or more entries are selected in a column.
doubleAction	The message sent to the target when an entry in the NXBrowser is double-clicked.
matrixClass	The class used to instantiate the matrices in the columns of NXBrowser; Matrix by default.

cellPrototype	A Cell that is copied to create new Cells in the NXBrowser's Matrices; NXBrowserCell by default.
pathSeparator	The character which separates the substrings of a path (see **getPath:ToColumn:**, **setPath:**).

Method Types

Initializing and freeing an NXBrowser	– initFrame: – free
Setting the delegate	– setDelegate: – delegate
Target and action	– setAction: – action – setTarget: – target – setDoubleAction: – doubleAction – sendAction
Setting component classes	– setMatrixClass: – setCellClass: – setCellPrototype: – cellPrototype
Setting NXBrowser behavior	– setMultipleSelectionEnabled: – isMultipleSelectionEnabled – setBranchSelectionEnabled: – isBranchSelectionEnabled – setEmptySelectionEnabled: – isEmptySelectionEnabled – reuseColumns: – setEnabled: – acceptsFirstResponder – acceptArrowKeys:andSendActionMessages: – getTitleFromPreviousColumn:

Configuring controls
- useScrollBars:
- useScrollButtons:
- setHorizontalScrollButtonsEnabled:
- areHorizontalScrollButtonsEnabled
- setHorizontalScrollerEnabled:
- isHorizontalScrollerEnabled

Setting the NXBrowser's appearance
- setMinColumnWidth:
- minColumnWidth
- setMaxVisibleColumns:
- maxVisibleColumns
- numVisibleColumns
- firstVisibleColumn
- lastVisibleColumn
- lastColumn
- separateColumns:
- columnsAreSeparated

Manipulating columns
- loadColumnZero
- isLoaded
- addColumn
- reloadColumn:
- displayColumn:
- displayAllColumns
- setLastColumn:
- selectAll:
- selectedColumn
- columnOf:
- validateVisibleColumns

Manipulating column titles
- setTitled:
- isTitled
- setTitle:ofColumn:
- titleOfColumn:
- getTitleFrame:ofColumn:
- titleHeight
- drawTitle:inRect:ofColumn:
- clearTitleInRect:ofColumn:

Scrolling an NXBrowser	– scrollColumnsLeftBy: – scrollColumnsRightBy: – scrollColumnToVisible: – scrollUpOrDown: – scrollViaScroller: – reflectScroll: – updateScroller
Event handling	– mouseDown: – keyDown: – doClick: – doDoubleClick:
Getting Matrices and Cells	– getLoadedCellAtRow:inColumn: – matrixInColumn: – selectedCell – getSelectedCells:
Getting column frames	– getFrame:ofColumn: – getFrame:ofInsideOfColumn:
Manipulating paths	– setPathSeparator: – setPath: – getPath:toColumn:
Drawing	– drawSelf::
Resizing the NXBrowser	– sizeTo:: – sizeToFit
Arranging an NXBrowser's components	– tile

Instance Methods

acceptArrowKeys:andSendActionMessages:

– **acceptArrowKeys:**(BOOL)*acceptFlag*
andSendActionMessages:(BOOL)*sendFlag*

Sets NXBrowser handling of arrow key input. If *acceptFlag* is YES, then the keyboard arrow keys move the selection whenever the NXBrowser or one of its subviews is the first responder; if *acceptFlag* is NO, arrow key input has no effect. Further, if *sendFlag* is YES, then when an arrow key is pressed, the NXBrowser's action message is sent as though the user had clicked on the new selection; if *sendFlag* is NO, then arrow keys only move the selection (if they are enabled). Returns **self**.

This method replaces the **acceptArrowKeys:** method from NeXTSTEP Release 2.

See also: – **acceptsFirstResponder**

acceptsFirstResponder

– (BOOL)**acceptsFirstResponder**

Returns YES if the NXBrowser accepts arrow key input, NO otherwise. The default setting is NO.

See also: – **acceptArrowKeys:andSendActionMessages:**

action

– (SEL)**action**

Returns the action sent to the target by the NXBrowser when the user makes a selection in one of its columns.

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

addColumn

– **addColumn**

Adds a column to the right of the last column in the NXBrowser and, if necessary, scrolls the NXBrowser so that the new column is visible. Your code should never invoke this method; it's invoked as needed by **doClick:** and **keyDown:** when the user selects a single branch node entry in the NXBrowser, and by s**etPath:** when it matches a path substring with a branch node entry. Override this method if you need the NXBrowser to do any additional updating when a column is added, but be sure to send this message to **super**. Returns **self**.

See also: – **loadColumnZero**, – **setPath:**, – **reloadColumn:**

areHorizontalScrollButtonsEnabled

– (BOOL)**areHorizontalScrollButtonsEnabled**

Returns YES if horizontal scroll buttons are used by the NXBrowser and are enabled, NO otherwise.

See also: – setHorizontalScrollButtonsEnabled:, – isHorizontalScrollerEnabled, – setHorizontalScrollerEnabled:

cellPrototype

– cellPrototype

Returns the NXBrowser's prototype Cell. This Cell is copied to create new Cells in the columns of the NXBrowser.

See also: – setCellPrototype:

clearTitleInRect:ofColumn:

– **clearTitleInRect:**(const NXRect *)*aRect* **ofColumn:**(int)*column*

Clears the title displayed in *aRect* above *column.* Your code shouldn't invoke this method directly; it's sent whenever a column title needs to be cleared. You can override this method if you draw your own column titles. *aRect* is in the NXBrowser's coordinate system. Returns **self**.

columnOf:

– (int)**columnOf:***matrix*

Returns the index of the column containing *matrix*; the leftmost (root) column is 0. Returns –1 if no column contains *matrix.*

See also: – matrixInColumn:

columnsAreSeparated

– (BOOL)**columnsAreSeparated**

Returns YES if columns are separated by a bezeled bar; NO if they're separated by a black line. If the NXBrowser is set to display column titles, its columns are automatically separated by bezels.

See also: – **separateColumns:**, – **setTitled:**

delegate

– **delegate**

Returns the delegate of the NXBrowser, the object that provides data for the NXBrowser and responds to certain notification messages.

See also: – **setDelegate:**

displayAllColumns

– **displayAllColumns**

Causes all columns currently visible in the NXBrowser to be redisplayed. This method is useful for redisplaying the NXBrowser after manipulating it with display disabled in the window (for instance, if Cells in some of the columns are deleted). Returns **self**.

See also: – **displayColumn:**

displayColumn:

– **displayColumn:**(int)*column*

Validates and displays column number *column*. *column* must already be loaded. This method is useful for updating the NXBrowser after manipulating *column* with display disabled in the window. Returns **self**.

See also: – **displayAllColumns**

doClick:

– **doClick:***sender*

Your code should never invoke this method. This is the action message sent to the NXBrowser by a column's Matrix when a mouse-down event occurs in a column. It sets the NXBrowser's last column to that of the Matrix where the click occurred, and removes any columns to the right that were previously loaded in the NXBrowser. If a single branch node entry is selected by the event, this method sends **addColumn** to **self** to display the corresponding data in the column to the right. It also sends the NXBrowser's action message to its target. Returns **self**.

You may want to override this method to add specific behavior for mouse clicks.

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

doDoubleClick:

– **doDoubleClick:***sender*

Your code should never invoke this method. This is the action message sent to the NXBrowser by a column's Matrix when a double-click occurs in a column. This method simply sends the double-click action message to the target; if no double-click action message is set, it sends the regular (single-click) action. Returns **self**.

You may want to override this method to add specific behavior for double-click events.

See also: – **doubleAction**, – **target**, – **doClick:**

doubleAction

– (SEL)**doubleAction**

Returns the action sent by the NXBrowser to its target when the user double-clicks an entry. If no double-click action message has been set, this method returns the regular (single-click) action.

See also: – **setDoubleAction:**, – **action**, – **target**, – **doDoubleClick:**

drawSelf::

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

Draws the NXBrowser, loading column 0 if it has not been loaded. Override this method if you change the way NXBrowser draws itself. Your code should never invoke this method; it's invoked by the **display** method. Returns **self**.

See also: – **display** (View)

drawTitle:inRect:ofColumn:

– **drawTitle:**(const char *)*title*
inRect:(const NXRect *)*aRect*
ofColumn:(int)*column*

Your code should never invoke this method. It's invoked whenever the NXBrowser needs to draw a column title. You may override it if you want your own column titles drawn. Returns **self**.

firstVisibleColumn

– (int)**firstVisibleColumn**

Returns the index of the leftmost visible column.

See also: – **lastVisibleColumn**

free

– **free**

Frees the NXBrowser and all the objects it manages: ScrollViews, Matrices, Cells, scroll Buttons, prototypes, and so on. Returns **nil**.

getFrame:ofColumn:

– (NXRect *)**getFrame:**(NXRect *)*theRect* **ofColumn:**(int)*column*

Returns a pointer to the rectangle (in NXBrowser coordinates) containing *column*; the pointer is returned both explicitly by the method and by reference in *theRect*. The returned rectangle includes the bezel area surrounding the column. If column isn't currently loaded

or displayed, this method returns NULL, without changing the coordinates of the rectangle represented in *theRect*. It also returns NULL if *theRect* is NULL.

See also: – **getFrame:ofInsideOfColumn:**

getFrame:ofInsideOfColumn:

– (NXRect *)**getFrame:**(NXRect *)*theRect* **ofInsideOfColumn:**(int)*column*

Returns a pointer to the rectangle (in NXBrowser coordinates) containing the "inside" of column number *column*; the pointer is returned both explicitly by the method and by reference in *theRect*. The "inside" is defined as the area in the column that contains the Cells and only that area (that is, no bezels). If *column* isn't currently loaded or displayed, this method returns NULL, without changing the coordinates of the rectangle represented in *theRect*. It also returns NULL if *theRect* is NULL.

See also: – **getFrame:ofColumn:**

getLoadedCellAtRow:inColumn:

– **getLoadedCellAtRow:**(int)*row* **inColumn:**(int)*column*

Returns the Cell at *row* in *column*, if that column is currently in the NXBrowser. This method creates and loads the Cell if necessary. It's the safest way to get a particular Cell in a column, since lazy delegates don't load every Cell in a Matrix and very lazy delegates don't even create all Cells until they're displayed. This method is preferred to the Matrix method **cellAt::**. If the specified *column* isn't in the NXBrowser, or if *row* doesn't exist in *column*, returns **nil**.

See also: – **browser:loadCellAtRow:inColumn:** (delegate method)

getPath:toColumn:

– (char *)**getPath:**(char *)*thePath* **toColumn:**(int)*column*

Returns a pointer to the string representing the path to *column*, both explicitly and by reference in *thePath*. Before invoking this method, you must allocate sufficient memory to accept the entire path string, and set *thePath* as a pointer to that memory. If *column* isn't loaded or *thePath* is a null pointer, this method returns NULL.

The path is constructed by concatenating the string values in the selected Cells in each column, preceding each with the path separator. For example, consider a path separator "@" and an NXBrowser with two columns. If the selected Cell in the left column has the string value "fowl" and the selected Cell in the right column has the string value "duck,"

the resulting path is "@fowl@duck." If there is no selection in the NXBrowser, the path will be "" (an empty string—not "@"). The default pathSeparator is the slash character ("/").

If multiple selection is enabled and there are multiple Cells selected, this method returns the path of the last (that is, the lowest) Cell in the last column. Once the path is retrieved, the selected Cells can all be retrieved with **getSelectedCells:**, and their titles used with the path to manipulate the data they represent.

See also: – **getSelectedCells:**, – **pathSeparator**, – **setPath:**

getSelectedCells:

– **getSelectedCells:**(List *)*aList*

Fills the provided List with the Cells in the NXBrowser that are selected. The previous contents of *aList* are not altered; your code should be sure *aList* is empty before invoking this method. If *aList* is **nil**, this method creates a List from the zone of the Matrix containing the Cells and returns it. Returns *aList*, or the created List.

See also: – **selectedCell**

getTitleFrame:ofColumn:

– (NXRect *)**getTitleFrame:**(NXRect *)*theRect* **ofColumn:**(int)*column*

Returns the rectangle (in NXBrowser coordinates) enclosing the title of column number *column*, both explicitly and by reference in *theRect*. If the NXBrowser isn't displaying titles or if the specified column isn't loaded, returns this method returns NULL.

getTitleFromPreviousColumn:

– **getTitleFromPreviousColumn:**(BOOL)*flag*

If *flag* is YES, sets the NXBrowser so that each column takes its title from the string value in the selected Cell in the column to its left, leaving column 0 untitled; use **setTitle:ofColumn:** to give column 0 a title. This method affects the receiver only when it is titled (that is, when **isTitled** returns YES).

By default, the NXBrowser is set to get column titles from the previous column. Send this message with NO as the argument if your delegate implements the

browser:titleOfColumn: method or if you use the **setTitle:ofColumn:** method to set all column titles. Returns **self**.

See also: – **setTitled:**, – **setTitle:ofColumn:**, – **browser:titleOfColumn:** (delegate method)

initFrame:

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

Initializes a new instance of NXBrowser with a bounds of *frameRect*. The initialized NXBrowser is set to have column titles, to get titles from previous columns, and to use scrollbars. The minimum column width is set to 100 and the path separator is set to the slash ("/") character. The NXBrowser is set not to clip. This method invokes the **tile** method to arrange the components of the NXBrowser (titles, scroll bars, Matrices, and so on).

isBranchSelectionEnabled

– (BOOL)**isBranchSelectionEnabled**

Returns YES if a branch node can be selected, NO otherwise.

See also: – **setBranchSelectionEnabled:**

isEmptySelectionEnabled

– (BOOL)**isEmptySelectionEnabled**

Returns YES if it is possible to have no Cells selected, NO otherwise.

See also: – **setEmptySelectionEnabled:**

isHorizontalScrollerEnabled

– (BOOL)**isHorizontalScrollerEnabled**

Returns YES if the NXBrowser uses a horizontal Scroller (instead of Buttons), and if that Scroller is enabled, NO otherwise.

See also: – **setHorizontalScrollerEnabled:**, – **setHorizontalScrollButtonsEnabled:**

isMultipleSelectionEnabled

– (BOOL)**isMultipleSelectionEnabled**

Returns YES if more than one leaf node can be selected, NO otherwise.

See also: – **setMultipleSelectionEnabled:**

isLoaded

– (BOOL)**isLoaded**

Returns YES if any of the NXBrowser's columns are loaded.

See also: – **loadColumnZero**, – **setPath:**

isTitled

– (BOOL)**isTitled**

Returns YES if the NXBrowser's columns are displayed with titles above them; NO otherwise.

See also: – **getTitleFromPreviousColumn:**, – **setTitled:**

keyDown:

– **keyDown:**(NXEvent *)*theEvent*

Handles arrow key events. This method is invoked when the NXBrowser or one of its subviews is the first responder. If the NXBrowser has been set to accept arrow keys, and the key represented in *theEvent* is an arrow key, this method scrolls through the NXBrowser in the direction indicated.

See also: – **acceptArrowKeys:andSendActionMessages:**, – **acceptsFirstResponder**, – **mouseDown:**

lastColumn

– (int)**lastColumn**

Returns the index of the last loaded column in the NXBrowser.

See also: – **lastVisibleColumn**

lastVisibleColumn

– (int)**lastVisibleColumn**

Returns the index of the rightmost visible column. This may be less than the value returned by **lastColumn** if the NXBrowser has been scrolled left.

See also: – **firstVisibleColumn**, – **lastColumn**

loadColumnZero

– **loadColumnZero**

Loads and displays data in column 0 of the NXBrowser, unloading any columns to the right that were previously loaded. Invoke this method to force the NXBrowser to be loaded; for example, after initializing the NXBrowser, when changing delegates, or when changing the data set managed by the delegate You may want to override this method if you subclass NXBrowser.

See also: – **setPath:**, – **reloadColumn:**

matrixInColumn:

– **matrixInColumn:**(int)*column*

Returns the Matrix found in column number *column*. Returns **nil** if column number *column* isn't loaded in the NXBrowser.

maxVisibleColumns

– (int)**maxVisibleColumns**

Returns the maximum number of visible columns allowed. No matter how many loaded columns the NXBrowser contains, or how large the NXBrowser is made (for example, by resizing its window), it will never display more than this number of columns. If the number of loaded columns can exceed the value returned by this method, the NXBrowser must display left and right scroll buttons.

See also: – **setMaxVisibleColumns:**, – **numVisibleColumns**,
– **setHorizontalScrollerEnabled:**, – **setHorizontalScrollButtonsEnabled:**

minColumnWidth

– (int)**minColumnWidth**

Returns the minimum width of a column in PostScript points (rounded to the nearest integer). No column will be smaller than the returned value unless the NXBrowser itself is smaller than that. The default setting is 100 points.

See also: – **setMinColumnWidth:**

mouseDown:

– **mouseDown:**(NXEvent *)*theEvent*

Handles a mouse down in the NXBrowser's left or right scroll buttons. Override this method if you need to to any special event processing. Returns **self**.

See also: – **keyDown:**

numVisibleColumns

– (int)**numVisibleColumns**

Returns the number of columns which can be visible at the same time in the NXBrowser (that is, the current width, in columns, of the NXBrowser). This may be less than the value returned by **maxVisibleColumns** if the window containing the NXBrowser has been resized.

See also: – **setMaxVisibleColumns:**, – **maxVisibleColumns:**

reflectScroll:

– **reflectScroll:***clipView*

This method updates scroll bars in the column containing *clipView*. Scroll bars are enabled if a column contains more data than can be displayed at once and disabled if the column can display all data. You should never need to invoke this method, but you may want to override it. Returns **self**.

See also: – **useScrollBars:**

reloadColumn:

– **reloadColumn:**(int)*column*

Reloads the previously loaded column number *column* by sending a message to the delegate to update the Cells in its Matrix, then reselecting the previously selected Cell if it's still in the Matrix. You should never send this message for a column that hasn't been loaded (you can check for this with the **lastColumn** method). Redraws the column and returns **self**.

See also: – **lastColumn**

reuseColumns:

– **reuseColumns:**(BOOL)*flag*

Sets whether the NXBrowser saves a column's Matrix and ClipView or ScrollView when the column is removed, and whether it then reuses these subviews when the column is reloaded. If *flag* is YES, the NXBrowser reuses columns for somewhat faster display of columns as they are reloaded. If *flag* is NO, the NXBrowser frees columns as they're unloaded, reducing average memory use. Returns **self**.

scrollColumnsLeftBy:

– **scrollColumnsLeftBy:**(int)*shiftAmount*

Scrolls the columns in the NXBrowser left by *shiftAmount* columns, making higher numbered columns visible. If *shiftAmount* exceeds the number of loaded columns to the right of the first visible column, then the columns scroll left to make the last loaded column visible. Redraws and returns **self**.

See also: – **scrollColumnsRightBy:**

scrollColumnsRightBy:

– **scrollColumnsRightBy:**(int)*shiftAmount*

Scrolls the columns in the NXBrowser right by *shiftAmount* columns, making lower numbered columns visible. If *shiftAmount* exceeds the number of columns to the left of the first visible column, then the columns scroll right until column 0 is visible. Redraws and returns **self**.

See also: – **scrollColumnsLeftBy:**

scrollColumnToVisible:

– **scrollColumnToVisible:**(int)*column*

Scrolls the NXBrowser to make column number *column* visible. If there's no *column* in the NXBrowser, this method scrolls to the right as far as possible. Redraws and returns **self**.

scrollUpOrDown:

– **scrollUpOrDown:***sender*

Scrolls a column up or down. Your code shouldn't send this message; NXBrowser receives it from a column's scroll buttons. You may want to override it to perform additional updating. Returns **self**.

scrollViaScroller:

– **scrollViaScroller:***sender*

Scrolls the NXBrowser's columns left or right based on the position of the Scroller sending the message. Your code shouldn't send this message, but you may want to override it. Returns **self**.

selectAll:

– **selectAll:***sender*

Selects all entries in the last column loaded in the NXBrowser if multiple selection is allowed. Returns **self**.

See also: – **setMultipleSelectionEnabled:**

selectedCell

– **selectedCell**

If there is a selection, returns the last selected Cell. "Last," in this context, means furthest to the right and lowest in the column.

See also: – **getSelectedCells:**

selectedColumn

– (int)**selectedColumn**

Returns the column number of the rightmost column containing a selected Cell. This won't be the last column if the selected Cell isn't a leaf. Returns –1 if no column in the NXBrowser contains a selected Cell.

See also: – **lastColumn**

sendAction

– **sendAction**

Sends the NXBrowser's action to its target and returns **self**.

See also: – **sendAction:to:** (Control)

separateColumns:

– **separateColumns:**(BOOL)*flag*

If *flag* is YES, sets NXBrowser so that columns have bezeled borders separating them; if NO, the borders are removed. When titles are set to display (by **setTitled:**), columns are automatically separated. Redraws the NXBrowser and returns **self**.

See also: – **setTitled:**

setAction:

– **setAction:**(SEL)*aSelector*

Sets the action of the NXBrowser. *aSelector* is the selector for the message sent to the NXBrowser's target when a mouse-down event occurs in a column of the NXBrowser. Returns **self**.

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

setBranchSelectionEnabled:

– **setBranchSelectionEnabled:**(BOOL)*flag*

Sets whether the user can select multiple branch and leaf node entries. If *flag* is YES and multiple selection is enabled (by **setMultipleSelectionEnabled:**), then multiple branch and leaf node entries can be selected. By default, a user can choose only multiple leaf node entries when multiple entry selection is enabled. Returns **self**.

This method replaces the **allowBranchSel:** method from NeXTSTEP Release 2.

See also: – **isBranchSelectionEnabled**, – **setMultipleSelectionEnabled:**

setCellClass:

– **setCellClass:***classId*

Sets the class of Cell used when adding Cells to a Matrix in a column of the NXBrowser. *classId* must be the value returned when sending the **class** message to NXBrowserCell or a subclass of NXBrowserCell. Since an NXBrowser always has its Matrices copy prototype Cells, this method simply makes a prototype, sends it an **init** message, and records that prototype. Returns **self**.

You shouldn't use Control's class method **setCellClass:** with an NXBrowser.

See also: – **setCellPrototype:**

setCellPrototype:

– **setCellPrototype:***aCell*

Sets *aCell* as the Cell prototype copied when adding Cells to the Matrices in the columns of NXBrowser. *aCell* must be an instance of NXBrowserCell or a subclass of NXBrowserCell. Each Matrix gets its own copy of *aCell* to use as a prototype, and will free that copy when the Matrix is freed. Returns the previous Cell prototype if *aCell* is an instance of either NXBrowserCell or a subclass of NXBrowserCell; otherwise, returns **nil**.

You shouldn't use Control's class method **setCellClass:** with an NXBrowser.

See also: – **cellPrototype**, – **setCellClass:**

setDelegate:

– **setDelegate:***anObject*

Sets the delegate of the NXBrowser to *anObject* and returns **self**. If *anObject* is of a class that implements the methods indicated in the class description for a normal, lazy, or very lazy delegate, it's set as the NXBrowser's delegate; otherwise, the delegate is set to **nil**. Returns **self**.

See also: – **delegate**

setDoubleAction:

– **setDoubleAction:**(SEL)*aSelector*

Sets the double-click action of the NXBrowser. *aSelector* is the selector for the action message sent to the target when a double-click occurs in one of the columns of the NXBrowser. Returns **self**.

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

setEmptySelectionEnabled:

– **setEmptySelectionEnabled:**(BOOL)*flag*

If *flag* is YES, the NXBrowser can display without any Cells selected; if *flag* is NO, then there must always be at least one Cell selected. By default, the setting is NO, and NXBrowser selects the first item in the first column. Returns **self**.

See also: – **isEmptySelectionEnabled**, – **setMultipleSelectionEnabled:**

setEnabled:

– **setEnabled:**(BOOL)*flag*

Enables the NXBrowser when *flag* is YES; disables it when *flag* is NO. Returns **self**.

See also: – **isEnabled** (Control)

setHorizontalScrollButtonsEnabled:

– **setHorizontalScrollButtonsEnabled:**(BOOL)*flag*

If *flag* is YES, sets the NXBrowser to use left and right scroll buttons. Generally, you should allow your NXBrowser to scroll horizontally unless your data is nonhierarchical, thus limited to a single column, or restricted so that the NXBrowser will always display enough columns for all data. Returns **self**.

This method, along with **setHorizontalScrollerEnabled:**, replaces the **hideLeftAndRightScrollButtons:** method from NeXTSTEP Release 2. In order to completely remove horizontal scrolling Controls, invoke both methods with arguments of NO.

See also: – **setHorizontalScrollerEnabled:**, – **areHorizontalScrollButtonsEnabled**

setHorizontalScrollerEnabled:

– **setHorizontalScrollerEnabled:**(BOOL)*flag*

If *flag* is YES, sets the NXBrowser to use a horizontal Scroller. Generally, you should allow your NXBrowser to scroll horizontally unless your data is nonhierarchical, thus limited to a single column, or restricted so that the NXBrowser will always display enough columns for all data. Returns **self**.

This method, along with **setHorizontalScrollButtonsEnabled:**, replaces the **hideLeftAndRightScrollButtons:** method from NeXTSTEP Release 2. In order to completely remove horizontal scrolling Controls, invoke both methods with arguments of NO.

See also: – **setHorizontalScrollButtonsEnabled:**,
– **areHorizontalScrollButtonsEnabled**

setLastColumn:

– **setLastColumn:**(int)*column*

Makes column number *column* the last column loaded and displayed by the NXBrowser. Removes any columns to the right of *column* from the NXBrowser, and scrolls columns in the NXBrowser to make the new last column visible if it wasn't previously. If column number *column* isn't already loaded, this method does nothing. Returns **self**.

See also: – **lastColumn**

setMatrixClass:

– **setMatrixClass:***classId*

Sets the class of Matrix used when adding new columns to the NXBrowser. *classId* must be the value returned by sending the **class** message to Matrix or a subclass of Matrix; otherwise this method retains the previous setting for the NXBrowser's Matrix class. NXBrowser initializes the Matrix of a new column with the **initFrame:mode:prototype: numRows:numCols:** method. Returns **self**.

setMaxVisibleColumns:

– **setMaxVisibleColumns:**(int)*columnCount*

Sets the maximum number of columns that may be displayed by the NXBrowser. Returns **self**.

To set the number of columns displayed in a new NXBrowser, first send it a **setMinColumnWidth:** message with a small argument (1, for example) to ensure that the desired number of columns will fit in the NXBrowser's frame. Then invoke this method to set the number of columns you want your NXBrowser to display. The minimum column width may then be reestablished to its desired value.

See also: – **maxVisibleColumns**, – **setMinColumnWidth:**

setMinColumnWidth:

– **setMinColumnWidth:**(int)*columnWidth*

Sets the minimum width for each column to *columnWidth*. If the new minimum width is different from the previous one, this method also redisplays the NXBrowser with columns set to the new width. *columnWidth* is measured in PostScript points rounded to the nearest integer. The default setting is 100. Returns **self**.

See also: – **minColumnWidth**

setMultipleSelectionEnabled:

– **setMultipleSelectionEnabled:**(BOOL)*flag*

Sets whether the user can select multiple entries in a column. If *flag* is YES, the user can choose any number of leaf entries in a column (or leaf and branch entries in a column if enabled by **setBranchSelectionEnabled:**). By default, the user can choose just one entry in a column at a time. Returns **self**.

This method replaces the **allowMultiSel:** method from NeXTSTEP Release 2.

See also: – **isMultipleSelectionEnabled**, – **setBranchSelectionEnabled:**

setPath:

– **setPath:**(const char *)*aPath*

Parses *aPath*—a string consisting of one or more substrings separated by the path separator—and selects column entries in the NXBrowser that match the substrings. If the first character in *aPath* is the path separator, this method begins searching for matches in column 0; otherwise, it begins searching in the last column loaded. If no column is loaded, this method loads column 0 and begins the search there. While parsing the current substring, it tries to locate a matching entry in the search column. If it finds an exact match, this method selects that entry and moves to the next column (loading the column if necessary) to search for the next substring.

If this method finds a valid path (one in which each substring is matched by an entry in the corresponding column), it returns **self**. If it doesn't find an exact match on a substring, it stops parsing *aPath* and returns **nil**; however, column entries that it has already selected remain selected.

Your code should never try to set a path or select items by sending Cell selection messages to the Matrices in the NXBrowser's columns. This bypasses every mechanism that allows the NXBrowser to update its display and load columns and Cells properly.

See also: – **getPath:toColumn**, – **pathSeparator**, – **setPathSeparator**, – **browser:selectCell:inColumn:** (delegate method)

setPathSeparator:

– **setPathSeparator:**(unsigned short)*charCode*

Sets the character used as the path separator; the default is the slash character ("/"). Returns **self**.

See also: – **getPath:toColumn**, – **setPath:**

setTarget:

– **setTarget:***anObject*

Sets the target of the NXBrowser. This is the object that will receive action messages when the user clicks or double-clicks on items in the NXBrowser. Returns **self**.

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

setTitle:ofColumn:

– **setTitle:**(const char *)*aString* **ofColumn:**(int)*column*

Sets the title of column number *column* in the NXBrowser to *aString*. If column *column* isn't loaded, this method does nothing. Returns **self**.

See also: – **getTitleFromPreviousColumn:**, – **setTitled:**,
– **browser:TitleOfColumn:** (delegate method)

setTitled:

– **setTitled:**(BOOL)*flag*

If *flag* is YES, columns display titles and are separated by bezeled borders. Returns **self**.

See also: – **getTitleFromPreviousColumn:**, – **setTitle:ofColumn:**,
– **browser:TitleOfColumn:** (delegate method)

sizeTo::

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

Resizes the NXBrowser to the new *width* and *height*. Usually sent by the NXBrowser's Window. Override this method if your subclass of NXBrowser need to do any special updating when resized. Returns **self**.

sizeToFit

– **sizeToFit**

Resizes the NXBrowser to contain all the columns and Controls displayed in it. Returns **self**.

target

– **target**

Returns the target for the NXBrowser's action message. This object receives action messages when the user clicks or double-clicks on items in the NXBrowser.

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

tile

– **tile**

Arranges the various subviews of NXBrowser—Scrollers, columns, titles, and so on—without redrawing. Your code shouldn't send this message. It's invoked any time the appearance of the NXBrowser changes; for example, when scroll Buttons or scroll bars are set, a column is added, and so on. Override this method if your code changes the appearance of the NXBrowser (for example, if you draw your own titles above columns). Returns **self**.

titleHeight

– (NXCoord)**titleHeight**

Returns the height of titles drawn above the columns of the NXBrowser. Override this method if you display your own titles above the NXBrowser's columns.

titleOfColumn:

– (const char *)**titleOfColumn:**(int)*column*

Returns a pointer to the title string displayed above column number *column*. If that column isn't loaded in the NXBrowser, returns NULL.

updateScroller

– **updateScroller**

Updates the horizontal Scroller to reflect the position of the visible columns of the NXBrowser. Returns **self**.

useScrollBars:

– **useScrollBars:**(BOOL)*flag*

Sets NXBrowser to use scroll bars for its columns based on *flag*. If *flag* is YES and the NXBrowser was previously set to use only scroll Buttons, this method causes the scroll Buttons to display at the base of the scroll bars. By default, NXBrowser uses scroll bars without Buttons; send **useScrollButtons:** to also have Buttons at the bottoms of the Scrollers. This method redraws the NXBrowser. Returns **self**.

See also: – **useScrollButtons:**

useScrollButtons:

– **useScrollButtons:**(BOOL)*flag*

Sets the NXBrowser to use scroll Buttons for its columns based on *flag*. If *flag* is YES and the NXBrowser was previously set to use scroll bars, this method causes scroll buttons to display at the base of the scroll bars; otherwise the Buttons are displayed underneath each column. To get only Buttons under the columns, send **useScrollBars:** with NO as the argument, then send this message with YES as the argument. This method redraws the NXBrowser. Returns **self**.

See also: – **useScrollBars:**

validateVisibleColumns

– **validateVisibleColumns**

Validates the columns visible in the NXBrowser by invoking the delegate method **browser:columnIsValid:** for all visible columns. Use this method to confirm that the entries displayed in each visible column are valid before redrawing.

See also: – **browser:columnIsValid:** (delegate method)

Methods Implemented by the Delegate

browser:columnIsValid:

– (BOOL)**browser:***sender* **columnIsValid:**(int)*column*

This method is invoked by NXBrowser's **validateVisibleColumns** method to determine whether the contents currently loaded in column number *column* need to be updated. This is useful for data sets that may change over time, such as files in a file system, or lists from a shared set of data that others can change. Returns YES if the contents are valid; NO otherwise.

See also: – **browser:selectCell:inColumn:** (delegate method)

browserDidScroll:

– **browserDidScroll:***sender*

Notifies the delegate when the browser has finished scrolling horizontally. This can be useful for aligning other user interface items with the columns of the browser (for example, an icon path or a series of pop-up lists). Returns **self**.

See also: – **browserDidScroll:** (delegate method)

browser:fillMatrix:inColumn:

– (int)**browser:***sender*
fillMatrix:*matrix*
inColumn:(int)*column*

Invoked by the NXBrowser to query a normal or lazy browser for the contents of *column*. This method should create NXBrowserCells by sending **addRow** or **insertRowAt:** messages to *matrix*. The NXBrowser will resize them to fit in the Matrix—you can't control the size of an NXBrowserCell. Returns the number of items in *column*.

A normal delegate should create each NXBrowserCell and send them the messages **setLoaded:** and **setLeaf:**, and **setEnabled:** if necessary. A lazy delegate marks Cells as loaded only when they are about to be displayed; however, it may create and partially fill in information (such as the title), saving only the time-consuming operations for an actual request to load an individual Cell.

If you implement this method, don't implement the delegate method **browser:getNumRowsInColumn:**.

See also: – **browser:loadCell:atRow:inColumn:** (delegate method)

browser:getNumRowsInColumn:

– (int)**browser:***sender* **getNumRowsInColumn:**(int)*column*

Implemented by very lazy delegates, this method is invoked by the NXBrowser to ask the delegate for the number of rows in column number *column*. This method allows the NXBrowser to resize its scroll bar for a column without loading all the Cells in that column. Returns the number of rows in *column*.

If you implement this method, don't implement the delegate method **browser:fillMatrix:inColumn:**.

See also: – **browser:loadCell:atRow:inColumn:** (delegate method)

browser:loadCell:atRow:inColumn:

– **browser:***sender*
loadCell:*cell*
atRow:(int)*row*
inColumn:(int)*column*

Implemented by lazy and very lazy delegates. This method loads the entry in the provided NXBrowserCell *cell* for the specified row and column in the NXBrowser. The NXBrowser will resize the Cell to fit in the Matrix—you can't control the size of an NXBrowserCell. Returns **self**.

A lazy delegate should send a **setLoaded:** message to *cell*, as well as **setLeaf:**, **setStringValue:** and **setEnabled:** messages if needed. A very lazy delegate should send **setLoaded:**, **setLeaf:**, and **setStringValue:** messages to *cell*, and **setEnabled:** if needed.

See also: – **browser:fillMatrix:inColumn:** (delegate method),
– **browser:getNumRowsInColumn:** (delegate method)

browser:selectCell:inColumn:

– (BOOL)**browser:***sender*
selectCell:(const char *)*entry*
inColumn:(int)*column*

Asks NXBrowser's delegate to validate and select an entry in column number *column*. This method should load the Cell corresponding to *entry* if necessary, send it **setLoaded:** and **setLeaf:** messages as needed to indicate its state, and send the column's Matrix a **selectCell:** or **selectCellAt::** message to select that Cell. If there is no Cell corresponding to *entry*, the selection should be cleared by sending **selectCellAt::** to the Matrix with –1

and –1 as the arguments. Returns YES if the method successfully selects the Cell corresponding to *entry* in *column*; NO otherwise.

If the delegate doesn't implement this method, the NXBrowser searches for entries by scanning through the entire list of Cells in the column. This will always work properly for NXBrowsers that browse static data. However, if the data can change while the NXBrowser is in use (for example, if a new file is created or deleted), this method allows the delegate to find that new data and add it to the column, or to find out that it no longer exists (or that its status has changed) and mark it as disabled or remove the Cell altogether with Matrix's **removeRowAt:andFree:** method (be sure to free the Cell).

See also: – **browser:columnIsValid:** (delegate method), – **matrixInColumn:**, – **selectCell:** (Matrix), – **selectCellAt::** (Matrix), – **removeRowAt:andFree:** (Matrix)

browser:titleOfColumn:

– (const char *)**browser:***sender* **titleOfColumn:**(int)*column*

Invoked by NXBrowser to get the title for *column* from the delegate. This method is invoked if the delegate implements it, but only when the NXBrowser is titled and has received a **getTitleFromPreviousColumn:** message with NO as the argument. By default, the NXBrowser makes each column title the string value of the selected Cell in the previous column. Returns the string representing the title belonging above *column*.

See also: – **getTitleFromPreviousColumn:**, – **setTitle:ofColumn:**, – **setTitled:**

browserWillScroll:

– **browserWillScroll:***sender*

This method notifies the delegate when the browser is about to scroll horizontally. This can be useful for hiding other user interface items to prepare for aligning them with the columns of the browser (for example, an icon path or a series of pop-up lists). Returns **self**.

See also: – **browserDidScroll:** (delegate method)

NXBrowserCell

Inherits From: Cell : Object

Declared In: appkit/NXBrowserCell.h

Class Description

NXBrowserCell is the subclass of Cell used by default to display data in the column Matrices of NXBrowser. Many of NXBrowserCell's methods are designed to interact with NXBrowser and NXBrowser's delegate. The delegate implements methods for loading the Cells in NXBrowser by setting their values and status. If your code needs access to a specific NXBrowserCell, you can use the NXBrowser method **getLoadedCellAtRow:inColumn:**.

You may find it useful to create a subclass of NXBrowserCell to alter its behavior and to enable it to work with and display the type of data you wish to represent. Use NXBrowser's **setCellClass:** or **setCellPrototype:** methods to have it use your subclass.

See the NXBrowser class specification for more details. In particular, the "Methods Implemented by the Delegate" section describes how the NXBrowser's delegate interacts with both NXBrowser and NXBrowserCells.

Instance Variables

None declared in this class.

Method Types

Initializing an NXBrowserCell	– init – initTextCell:
Determining component sizes	– calcCellSize:inRect:
Accessing graphic attributes	– isOpaque + branchIcon + branchIconH
Displaying	– drawInside:inView: – drawSelf:inView: – highlight:inView:lit:
Placing in browser hierarchy	– setLeaf: – isLeaf
Determining loaded status	– setLoaded: – isLoaded
Setting state	– set – reset

Class Methods

branchIcon

+ branchIcon

Returns the NXImage object named "NXMenuArrow". This is the icon displayed to indicate a branch node in an NXBrowserCell. Override this method if you want your subclass to display a different branch icon.

See also: – isLeaf

branchIconH

+ branchIconH

Returns the NXImage object named "NXMenuArrowH". This is the highlighted icon displayed to indicate a selected branch node in an NXBrowserCell. Override this method if you want your subclass to display a different branch icon.

See also: – isLeaf

Instance Methods

calcCellSize:inRect:

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

Returns **self**, and, by reference in *theSize*, the minimum width and height required for displaying the NXBrowserCell in a given rectangle. Makes sure *theSize* remains large enough to accommodate the branch arrow icon. If it isn't possible for the NXBrowserCell to fit in *aRect*, the width or height returned in *theSize* could be bigger than those of the rectangle. Returns **self**.

drawInside:inView:

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

Draws the inside of the NXBrowserCell. Unlike other Cells, NXBrowserCell never draws a border or bezel. Override this method to draw the cell differently. Returns **self**.

See also: – **drawSelf:inView:**

drawSelf:inView:

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

Draws the inside of the NXBrowserCell by invoking **drawInside:inView:**. Unlike other Cells, NXBrowserCell never draws a border or bezel. Returns **self**.

See also: – **drawInside:inView:**

highlight:inView:lit:

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

If the highlighted state would change, sets the NXBrowserCell's highlighted state to *lit* and redraws it if needed within *cellFrame* in *aView*. Override this method to highlight the NXBrowserCell differently. Returns **self**.

See also: – **set**, – **reset**, – **highlight:inView:lit:** (Cell)

init

– **init**

Initializes and returns the receiver, a new NXBrowserCell instance, sets its title to "BrowserItem" and returns **self**.

See also: – **initTextCell:**

initTextCell:

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

Initializes the receiver, a new NXBrowserCell instance with *aString* as its title. Sets the NXBrowserCell so it doesn't wrap text. This method is the designated initializer for the NXBrowserCell class. Override this method if you create a subclass of NXBrowserCell that performs its own initialization. Returns **self**.

See also: – **init**

isLeaf

– (BOOL)**isLeaf**

Determines whether the entry in the receiver represents a leaf node (such as a file) or branch node (such as a directory). This method is invoked by NXBrowser to check whether to display the branch icon in the Cell and, when an NXBrowserCell is selected, whether to load a column to the right of the column containing the receiving Cell. Returns YES if the cell represents a leaf, NO if the cell represents a branch.

See also: – **setLeaf:**

isLoaded

– (BOOL)**isLoaded**

Returns YES if the NXBrowserCell is loaded, NO if it isn't. Used by NXBrowser to determine if a particular Cell is loaded in a column. When an NXBrowserCell is created, this value is YES; however, if the NXBrowserCell is created by the NXBrowser, it sets the value to NO so the delegate can properly set the loaded status. NXBrowser and its delegate change the value returned by this method using the **setLoaded:** method to reflect the current status of the cell.

See also: – **setLoaded:**, NXBrowser

isOpaque

– (BOOL)**isOpaque**

Returns YES since an NXBrowserCell is always opaque.

reset

– **reset**

Sets the NXBrowserCell's state to 0 and sets the highlighted flag to NO. Does not display the NXBrowserCell, even if autodisplay is on. Returns **self**.

See also: – **set**, – **highlight:inView:lit**

set

– **set**

Sets the NXBrowserCell's state to 1 and sets the highlighted flag to YES. Does not display the NXBrowserCell, even if autodisplay is on. Returns **self**.

See also: – **reset**, – **highlight:inView:lit**

setLeaf:

– **setLeaf:**(BOOL)*flag*

Invoked by NXBrowser's delegate when it loads an NXBrowserCell. If *flag* is YES, the NXBrowserCell is set to represent a leaf node; it will display without the branch icon. When *flag* is NO, the NXBrowserCell is set to represent a branch node; it will display with the branch icon. Does not display the NXBrowserCell, even if autodisplay is on. Returns **self**.

See also: – **isLeaf**, – **branchIcon**, – **branchIconH**

setLoaded:

– **setLoaded:**(BOOL)*flag*

Sets the loaded status of the NXBrowser cell to *flag*. This method is invoked by NXBrowser or its delegate to set the status of the NXBrowserCell. The delegate should send the **setLoaded:** message with YES as the argument when it loads the cell.

See also: – **isLoaded**, NXBrowser delegate methods

NXCachedImageRep

Inherits From: NXImageRep : Object

Declared In: appkit/NXCachedImageRep.h

Class Description

An NXCachedImageRep is a rendered image in a window, typically a window that stays off-screen. The only data that's available for reproducing the image is the image itself. Thus an NXCachedImageRep differs from the other kinds of NXImageReps defined in the Application Kit, all of which can reproduce an image from the information originally used to draw it.

Instances of this class are generally used indirectly, through an NXImage object. An NXCachedImageRep must be able to provide the NXImage with some information about the image—so that the NXImage can match it to a display device, for example, or know whether to scale it. Therefore, it's a good idea to use these inherited methods to inform the NXCachedImageRep object about the image in the cache:

setNumColors:
setAlpha:
setPixelsHigh:
setPixelsWide:
setBitsPerSample:

These methods are all defined in the NXImageRep class.

Instance Variables

None declared in this class.

Method Types

Initializing a new NXCachedImageRep	– initFromWindow:rect: – copyFromZone:
Freeing an NXCachedImageRep	– free
Getting the representation	– getWindow:andRect:
Drawing the image	– draw
Archiving	– read: – write:

Instance Methods

copyFromZone:

– **copyFromZone:**(NXZone *)*zone*

Returns a new instance of NXCachedImageRep that's an exact copy of the receiver. Memory for the new instance is allocated from *zone*. Cached image reps are copied fully, including their own copy of the image data.

See also: – **copyFromZone:** (NXImage)

draw

– (BOOL)**draw**

Reads image data from the cache and reproduces the image from that data. The reproduction is rendered in the current window at location (0.0, 0.0) in the current coordinate system.

It's much more efficient to reproduce an image by compositing it, which can be done through the NXImage class. An NXBitmapImageRep can also be used to reproduce an existing image.

This method returns YES if successful in reproducing the image, and NO if not.

See also: – **drawIn:** (NXImageRep), – **drawAt:** (NXImageRep), – **initData:fromRect:** (NXBitmapImageRep)

free

– **free**

Deallocates the NXCachedImageRep.

getWindow:andRect:

– **getWindow:**(Window **)*theWindow* **andRect:**(NXRect *)*theRect*

Copies the **id** of the Window object where the image is located into the variable referred to by *theWindow*, and copies the rectangle that bounds the image into the structure referred to by *theRect*. If *theRect* is NULL, only the Window **id** is provided. Returns **self**.

init

Generates an error message. This method cannot be used to initialize an NXCachedImageRep. Use the **initFromWindow:rect:** method instead.

See also: – **initFromWindow:rect:**

initFromWindow:rect:

– **initFromWindow:**(Window *)*aWindow* **rect:**(const NXRect *)*aRect*

Initializes the receiver, a new NXCachedImageRep instance, for an image that will be rendered within the *aRect* rectangle in *aWindow*, and returns the initialized object. The rectangle is specified in *aWindow*'s base coordinate system. The size of the image is set from the size of the rectangle.

You must draw the image in the rectangle yourself; there are no NXCachedImageRep methods for this purpose.

read:

– **read:**(NXTypedStream *)*stream*

Reads the NXCachedImageRep from the typed stream *stream*.

write:

– **write:**(NXTypedStream *)*stream*

Writes the NXCachedImageRep to the typed stream *stream*.

NXColorList

Inherits From: Object

Declared In: appkit/NXColorList.h

Class Description

Instances of NXColorList are used to manage named lists of NXColors. Colors are added to, looked up in, and removed from an NXColorList by name. NXColorPanel's list-mode color picker uses instances of NXColorList for the standard PANTONE® Colors, the NeXT colors, and any lists created by the user. An application can use NXColorList to manage document-specific color lists, which may be added to an application's NXColorPanel using its **attachColorList:** method.

An NXColorList manages colors in one of two ways. When managing a list such as the PANTONE Colors, an NXColorList issues colors with persistent names. When managing other color lists, such as the NeXT color list or a list created by the user, an NXColorList issues colors without names.

NXColors generated with persistent names reference both the color name and color list name. This reference remains even if the color is copied to another list and the user then renames the color in the NXColorPanel. Lists that generate colors with persistent names are considered immutable—attempts to change them at runtime will raise exceptions as described in the list editing methods. The PANTONE Color list is currently the only example of a list that issues colors with persistent names.

The Application Kit function **NXSetColor()** assures that colors with persistent names are printed correctly regardless of the list from which they are selected. Say, for example, the user has created a new color list in the color panel, copied a PANTONE Color into that list, given that color a new name, then used the color from the new list in a document. When that document is printed, the **NXSetColor()** function recognizes the color's persistent name and references the PANTONE Color list to find the correct color value for the device.

An NXColorList saves and retrieves its colors from files with the extension ".clr". The standard search path for the files for color lists include **~/Library/Colors**, **~/.Next/Colors** (for historical reasons), **/LocalLibrary/Colors**, and **/NextLibrary/Colors**. NXColorList reads color list files in several different—and undocumented—formats; NXColorList saves color lists using typed streams API.

The files for color lists such as the PANTONE Colors provided with NeXTSTEP are stored in file-wrappers. This allows for localized and device-dependent versions of such lists. Color lists created by the user from the NXColorPanel are saved as files in the directory **~/Library/Colors**. Color lists created for a document by an application may be saved in the document's file package.

See also: – **attachColorList:**, – **detachColorList:** (NXColorPanel)

Instance Variables

None declared in this class.

Method Types

Initializing and freeing	– init – initWithName: – initWithName:fromFile: – freeAndRemoveFile – free
Getting all color lists	+ availableColorLists
Color list names	+ findColorListNamed: – name
Managing colors by name	– setColorNamed:color: – colorNamed: – nameOfColorAt: – localizedNameForColorNamed: – removeColorNamed:
Generates persistent names	– generatesNamedColors
Editing	– isEditable
Number of colors	– colorCount
Saving to a file	– saveTo:
Archiving	– read: – write:

Class Methods

availableColorLists

+ (List *)**availableColorLists**

Returns a List object of all NXColorLists found in the standard color list directories. This list belongs to the NXColorList class and should not be cached by the caller. Color lists created at run time are not included in this list unless they were saved into one of the standard color list directories. This method is primarily for use by NXColorPanel.

The standard search path for color lists is discussed in the class description.

findColorListNamed:

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

Returns the list with the name *name*; if the list doesn't exist, returns **nil**. *name* mustn't include the ".clr" suffix. Color lists are searched for in the color list search path and among the named, registered lists. The search path for color lists is discussed in the class description.

See also: – **initWithName:**

Instance Methods

colorCount

– (unsigned)**colorCount**

Returns the number of colors in the NXColorList.

colorNamed:

– (NXColor)**colorNamed:**(const char *)*colorName*

Returns the NXColor associated with *colorName*. Raises the exception NX_colorUnknown if *colorName* isn't in the NXColorList.

See also: – **removeColorNamed:**, – **setColorNamed:color:**

free

– **free**

Frees the NXColorList and its storage.

freeAndRemoveFile

– **freeAndRemoveFile**

Frees the NXColorList. If the list belongs to the user, this method also deletes the file from which the list was created.

generatesNamedColors

– (BOOL)**generatesNamedColors**

Determines if the list generates colors with persistent names when its **colorNamed:** method is invoked. Colors with persistent names maintain references to their source list and color name, even if the user copies the color to another list and changes the name in the NXColorPanel. If an NXColorList generates colors with persistent names, this behavior can't be changed at runtime.

See also: – **colorNamed:**, – **removeColorNamed:**, – **setColorNamed:color:**

init

– **init**

Creates an unnamed NXColorList.

See also: – **initWithName:**, – **initWithName:fromFile:**

initWithName:

– **initWithName:**(const char *)*name*

Creates and returns a new NXColorList. Also registers it under the specified name if the name isn't in use already.

See also: – **initWithName:fromFile:**, – **saveTo:**

initWithName:fromFile:

– **initWithName:**(const char *)*name* **fromFile:**(const char *)*path*

Creates and returns a new NXColorList. *path* should be the full path to the file for the color list; *name* should be the name of the file for the color list (minus the ".clr" extension). This method also registers the NXColorList under *name* if that name isn't already in use. This method is the designated initializer for NXColorList.

See also: – **initWithName:**, – **saveTo:**

isEditable

– (BOOL)**isEditable**

Returns YES if the list doesn't generate colors with persistent names and if the user has both read and write access to the file from which the NXColorList was initiated; returns NO otherwise.

See also: – **generatesNamedColors**, – **initWithName:fromFile:**

localizedNameForColorNamed:

– (const char *)**localizedNameForColorNamed:**(const char *)*colorName*

Returns a name in the user's language of choice for the color associated with *colorName*. The directory for a color list (such as PANTONE colors) should contain a ".lproj" directory for each supported language. This method searches in the directory of the user's chosen language for a ".strings" file containing translated color names.

This method raises an exception if *colorName* isn't in the list. If the color has no translation for the user's chosen language, then *colorName* is returned.

name

– (const char *)**name**

Returns the name of the NXColorList.

See also: – **initWithName:**, – **initWithName:fromFile:**

nameOfColorAt:

– (const char *)**nameOfColorAt:**(unsigned)*count*

Returns the name associated with the NXColor at *count* in the NXColorList.

read:

– **read:**(NXTypedStream *)*stream*

Reads the NXColorList from the specified stream. Returns **self**.

removeColorNamed:

– (void)**removeColorNamed:**(const char *)*colorName*

Removes the specified color. This method raises the exception NX_colorNotEditable if the NXColorList generates named colors; it raises the exception NX_colorUnknown if *colorName* isn't in the NXColorList.

See also: – generatesNamedColors, – isEditable

saveTo:

– **saveTo:**(const char *)*path*

If *path* is a directory, saves the NXColorList in a file named *listname*.clr (where *listname* is the name with which the NXColorList was initialized). If *path* includes a file name, this method saves the file under that name. If *path* isn't specified, then this method saves the file as **~/Library/Colors** in a file named *listname*.clr.

See also: – initWithName:, – initWithName:fromFile:

setColorNamed:color:

– (void)**setColorNamed:**(const char *)*colorName* **color:**(NXColor)*color*

Adds the NXColor *color* to the list and associates it with the name *colorName*. If *colorName* is already in the list, this method sets its NXColor to *color*. This method raises the exception NX_colorNotEditable if the NXColorList generates named colors; it raises the exception NX_colorUnknown if *colorName* isn't in the NXColorList.

write:

– **write:**(NXTypedStream *)*stream*

Writes the NXColorList to the specified stream. Returns **self**.

Methods Implemented by the Delegate

colorListDidChange:colorName:

– **colorListDidChange:***list* **colorName:**(const char *)*colorName*

Indicates that a color in the NXColorList was changed. This method is invoked when the list is first created or when the **setColorNamed:color:** or **removeColorNamed:** method is invoked.

See also: – **initWithName:fromFile:**, – **removeColorNamed:**, – **setColorNamed:color:**

NXColorPanel

Inherits From: Panel : Window : Responder : Object

Declared In: appkit/NXColorPanel.h

Class Description

NXColorPanel provides a standard user interface for selecting color in an application. It provides a number of standard color selection modes, and, with the NXColorPickingDefault and NXColorPickingCustom protocols, allows an application to add its own color selection modes. It allows the user to set swatches containing frequently used colors. Once set, these swatches are displayed by NXColorPanel in any application where it is used, giving the user color consistency between applications. NXColorPanel enables users to capture a color anywhere on the screen for use in the active application, and provides API for dragging colors between views in an application. NXColorPanel's action message is sent to the target object when the user changes the current color.

An application has only one instance of NXColorPanel, the shared instance. Invoking the **sharedInstance:** method returns the shared instance of NXColorPanel, instantiating it if necessary. You can also initialize an NXColorPanel for your application by invoking Application's **orderFrontColorPanel** method.

You can put NXColorPanel in any application created with Interface Builder by adding the "Colors..." item from the Menu palette to the application's menu.

Color Mask and Color Modes

The color mask determines which of the color modes are enabled for NXColorPanel. This mask is set before you initialize a new instance of NXColorPanel. NX_ALLMODESMASK represents the logical OR of the other color mask constants: it causes the NXColorPanel to display all standard color pickers. When initializing a new instance of NXColorPanel, you can logically OR any combination of color mask constants to restrict the available color modes.

Mode	Color Mask Constant
Grayscale-Alpha	NX_GRAYMODEMASK
Red-Green-Blue	NX_RGBMODEMASK
Cyan-Yellow-Magenta-Black	NX_CMYKMODEMASK
Hue-Saturation-Brightness	NX_HSBMODEMASK
TIFF image	NX_CUSTOMPALETTEMODEMASK
Custom color lists	NX_COLORLISTMODEMASK
Color wheel	NX_WHEELMODEMASK
All of the above	NX_ALLMODESMASK

The NXColorPanel's color mode mask is set using the class method **setPickerMask:.** The mask must be set before creating an application's instance of NXColorPanel.

When an application's instance of NXColorPanel is masked for more than one color mode, your program can set its mode by invoking the **setMode**: method with a color mode constant as its argument; the user can set the mode by clicking buttons on the panel. Here are the standard color modes and mode constants:

Mode	Color Mode Constant
Grayscale-Alpha	NX_GRAYMODE
Red-Green-Blue	NX_RGBMODE
Cyan-Yellow-Magenta-Black	NX_CMYKMODE
Hue-Saturation-Brightness	NX_HSBMODE
TIFF image	NX_CUSTOMPALETTEMODE
Color lists	NX_COLORLISTMODE
Color wheel	NX_BEGINMODE

In grayscale-alpha, red-green-blue, cyan-magenta-yellow-black, and hue-saturation-brightness modes, the user adjusts colors by manipulating sliders. In the custom palette mode, the user can load a TIFF file into the NXColorPanel, then select colors from the TIFF image. In custom color list mode, the user can create and load lists of named colors. The two custom modes provide PopUpLists for loading and saving files. Finally, color wheel mode provides a simplified control for selecting colors; by default, it's the initial mode when the NX_ALLMODESMASK constant is used to initialize the NXColorPanel.

Associated Classes and Protocols

The NXColorList class provides an API for managing custom color lists. The NXColorPanel methods **attachColorList:** and **detachColorList:** let your application add and remove custom lists from the NXColorPanel's user interface.

The protocols NXColorPickingDefault and NXColorPickingCustom provide an API for adding custom color selection to the user interface. The NXColorPicker class implements the NXColorPickingDefault protocol; you can subclass NXColorPicker and implement the NXColorPickingCustom protocol in your subclass to create your own user interface for color selection.

See also: NXColorList, NXColorPicker, NXColorPickingDefault protocol, NXColorPickingCustom protocol, NXColorWell

Instance Variables

None declared in this class.

Method Types

Creating a New NXColorPanel	+ sharedInstance:
Setting Color	– setColor: – color
Target and Action	– setAction: – setTarget: – setContinuous: – isContinuous
Mode	– setMode: – mode
Alpha	– alpha – setShowAlpha: – doesShowAlpha
Picker settings	+ setPickerMode: + setPickerMask:
Accessory View	– setAccessoryView: – accessoryView
Color list management	– attachColorList: – detachColorList: – updateCustomColorList
Color dragging	+ dragColor:withEvent:fromView:
Archiving	– read:

Class Methods

alloc

Generates an error message. This method cannot be used to create NXColorPanel instances. Use the **sharedInstance:** class method instead.

See also: + **sharedInstance:**

allocFromZone:

Generates an error message. This method cannot be used to create NXColorPanel instances. Use the **sharedInstance:** class method instead.

See also: + **sharedInstance:**

dragColor:withEvent:fromView:

+ (BOOL)**dragColor:**(NXColor)*color*
withEvent:(NXEvent *)*theEvent*
fromView:*controlView*

Drags colors between views in an application. This method is usually invoked by the **mouseDown:** method of *controlView*. The dragging mechanism handles all subsequent events.

Because it is a class method, **dragColor:withEvent:fromView:** can be invoked whether or not the instance of NXColorPanel exists. Returns YES.

setPickerMask:

+ (void)**setPickerMask:**(int)*mask*

Accepts as a parameter one or more logically OR'd color mode masks (defined in the header file **appkit/NXColorPanel.h**). This determines which color selection modes will be available in an application's NXColorPanel. This method only has an effect before NXColorPanel is instantiated.

If you create a class that implements the color picking protocols (NXColorPickingDefault and NXColorPickingCustom), you may want to give it a unique mask—one different from those defined for the standard color pickers. To display your color picker, your application

will need to logically OR that unique mask with the standard color mask constants when invoking this method.

See also: NXColorPicker class, NXColorPickingDefault protocol, NXColorPickingCustom protocol

setPickerMode:

+ (void)**setPickerMode:**(int)*mode*

Sets the color panel's initial picker mode. The mode determines which picker will initially be visible. This method may be called at any time, whether or not an application's NXColorPanel has been instantiated.

See also: – **setMode:**, – **setMode:** (NXColorPicker)

sharedInstance:

+ **sharedInstance:**(BOOL)*create*

Returns the shared instance of NXColorPanel. If *create* is YES, this method creates, if necessary, and returns the NXColorPanel. If *create* is NO and the shared instance exists, this method returns it; if no instance of NXColorPanel exists, returns **nil**.

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

Instance Methods

accessoryView

– **accessoryView**

Returns the NXColorPanel's accessory View.

See also: – **setAccessoryView:**

attachColorList:

– **attachColorList:**theColorList

Notifies color pickers (objects that conform to the NXColorPickingDefault and NXColorPickingCustom protocols) when a new NXColorList is added to the NXColorPanel. Your application should use this method to add an NXColorList saved with a document in its file package or in a directory other than NXColorList's standard search directories. This method invokes **attachColorList:** on all color pickers in the application.

See also: – **detachColorList:**, NXColorList, NXColorPicker, NXColorPickingDefault protocol, NXColorPickingCustom protocol

alpha

– (float)**alpha**

Returns the current alpha level of the NXColorPanel based on its opacity slider. If the NXColorPanel has no opacity slider, returns 1.0 (opaque).

See also: – **doesShowAlpha**, – **setShowAlpha:**

color

– (NXColor)**color**

Returns the color selected in the NXColorPanel.

See also: – **setColor**

detachColorList:

– **detachColorList:**theColorList

Notifies color pickers (objects that conform to the NXColorPickingDefault and NXColorPickingCustom protocols) when an NXColorList is removed from the NXColorPanel. Your application should use this method to remove an NXColorList saved with a document in its file package or in a directory other than NXColorList's standard search directories. This method invokes **detachColorList:** on all color pickers in the application.

See also: – **attachColorList:**, NXColorList, NXColorPicker, NXColorPickingDefault protocol, NXColorPickingCustom protocol

doesShowAlpha

– (BOOL)**doesShowAlpha**

Returns YES if the alpha (opacity) slider is currently displayed by the NXColorPanel; NO if not. The opacity slider is independent of the currently selected color picker.

See also: – **setShowAlpha**

isContinuous

– (BOOL)**isContinuous**

Returns whether or not the NXColorPanel's color is being set continuously as the user manipulates the color picker.

See also: – **setContinuous:**

mode

– (int)**mode**

Returns the current color picker mode for the NXColorPanel. The mode constants for the standard color pickers are listed in the class description.

read:

– **read:**(NXTypedStream *)*theStream*

Reads the NXColorPanel from the typed stream *theStream*. Returns **self**.

setAccessoryView:

– **setAccessoryView:***aView*

Sets the accessory View displayed in the NXColorPanel to *aView*. The accessory View can be any custom View that you want to display with NXColorPanel, such as a View offering color blends in a drawing program. The accessory View is displayed below the color picker and above the color swatches in the NXColorPanel. The NXColorPanel automatically resizes to accommodate the accessory View. Returns the previous accessory view, if there was one; otherwise, returns **nil**.

See also: – **accessoryView**

setAction:

– **setAction:**(SEL)*aSelector*

Sets the action of the NXColorPanel to *aSelector*. Returns **self**.

See also: – **setTarget:**

setColor:

– **setColor:**(NXColor)*color*

Sets the color setting of the NXColorPanel to *color* and redraws the panel. Returns **self**.

See also: – **color**

setContinuous:

– **setContinuous:**(BOOL)*flag*

Sets the NXColorPanel to send the action message to its target continuously as the color of the NXColorPanel is set by the user. Send this message with *flag* YES if, for example, you want to continuously update the color of the target. Returns **self**.

See also: – **isContinuous**

setMode:

– **setMode:**(int)*mode*

Sets the mode of the NXColorPanel if *mode* is one of the modes allowed by the color mask. The color mask is set when you first create the shared instance of NXColorPanel for an application. *mode* may be one of these symbolic constants, declared in the header file **appkit/NXColorPanel.h**:

NX_GRAYMODE
NX_RGBMODE
NX_CMYKMODE
NX_HSBMODE
NX_CUSTOMPALETTEMODE
NX_COLORLISTMODE
NX_WHEELMODE

If you create a color picker—a class that implements the NXColorPickingDefault and NXColorPickingCustom protocols—it should define a unique mode value that differs from those for the standard color pickers.

Color modes and masks are described in the class description.

Returns **self**.

See also: – mode

setShowAlpha:

– setShowAlpha:(BOOL)*flag*

If *flag* is YES, sets the NXColorPanel to show alpha. Returns **self**.

See also: – doesShowAlpha

setTarget:

– setTarget:*anObject*

Sets the target of the NXColorPanel to *anObject*. The NXColorPanel's target is the object to which the action message is sent when the user selects a color. Returns **self**.

See also: – setAction:, – setContinuous:

updateCustomColorList

– updateCustomColorList

Updates the current custom color lists. This method sends each color picker an **updateColorList:** message with a **nil** argument. Color pickers are objects conforming to the NXColorPickingDefault and NXColorPickingCustom protocols.

NXColorPicker

Inherits From: Object

Conforms To: NXColorPickingDefault

Declared In: appkit/NXColorPicker.h

Class Description

NXColorPicker is an abstract superclass that implements the NXColorPickingDefault protocol. The NXColorPickingDefault and NXColorPickingCustom protocols define a way to add color pickers—custom user interfaces for color selection—to the NXColorPanel. The simplest way to implement a color picker is to create a subclass of NXColorPicker that implements the NXColorPickingCustom protocol.

The NXColorPickingDefault protocol specification describes the details of implementing a color picker and adding it to your application's NXColorPanel; you should turn there first for an overview of how NXColorPicker works. This specification is provided to document the specific behavior of NXColorPicker's methods.

Adopted Protocols

NXColorPickingDefault
- initFromPickerMask:withColorPanel:
- provideNewButtonImage
- insertNewButtonImage:in:
- viewSizeChanged:
- alphaControlAddedOrRemoved:
- insertionOrder
- attachColorList:
- detachColorList:
- updateColorList:
- setMode:

Instance Variables

id **imageObject**
NXColorPanel ***colorPanel**
BOOL **continuous**

imageObject	Object providing the ButtonCell image
colorPanel	Panel in which the color picker is installed
continuous	YES if color picker updates current color continuously

Instance Methods

alphaControlAddedOrRemoved:

– **alphaControlAddedOrRemoved:***sender*

Does nothing and returns **self**.

attachColorList:

– **attachColorList:***colorList*

Does nothing and returns **self**.

detachColorList:

– **detachColorList:***colorList*

Does nothing and returns **self**.

insertionOrder

– (float)**insertionOrder**

Returns 0.4, a value that places NXColorPicker's ButtonCell first in the Matrix from which the user selects color pickers.

See also: – **insertNewButtonImage:in:**, – **provideNewButtonImage:**

insertNewButtonImage:in:

– **insertNewButtonImage:***newImage* **in:***newButtonCell*

Sets *newImage* as *newButtonCell*'s image by invoking ButtonCell's **setImage:** method. Returns **self**.

initFromPickerMask:withColorPanel:

– **initFromPickerMask:***(int)theMask* **withColorPanel:***thePanel*

Sets the color picker's color panel to *thePanel* and returns **self**. Override this method to respond to the values in *theMask* or do other custom initialization. If you override this method in a subclass, you should forward the message to **super** as part of the implementation.

provideNewButtonImage

– **provideNewButtonImage**

Returns the NXImage that represents the NXColorPicker in the NXColorPanel's Matrix of ButtonCells. This method attempts to load the image from a file named *MyPickerClass*.**tiff** (where *MyPickerClass* is the name of your subclass of NXColorPicker) in the *MyPickerClass* bundle of the application's ColorPicker directory. See "Color Picker Bundles" in the NXColorPickingDefault protocol specification for a more complete discussion of this bundle.

setMode:

– **setMode:**(int)*mode*

Does nothing and returns **self**. Override this method if your color picker has submodes to set the mode of the color picker to *mode*.

updateColorList:

– **updateColorList:***colorList*

Does nothing and returns **self**.

viewSizeChanged:

– **viewSizeChanged:***sender*

Does nothing and returns **self**.

NXColorWell

Inherits From: Control : View : Responder : Object

Declared In: appkit/NXColorWell.h

Class Description

NXColorWell is a Control for selecting and displaying a single color value. An example of NXColorWell is found in NXColorPanel, which uses a well to display the current color selection. NXColorWell is available from the Palettes panel of Interface Builder.

An application can have one or more active NXColorWells. You can activate multiple NXColorWells by invoking the **activate:** method with NO as its argument. You can set the same color for all active color wells by invoking the class method **activeWellsTakeColorFrom:**. You can deactivate multiple wells using the class method **deactivateAllWells**. When a mouse-down event occurs in an NXColorWell, it becomes the only active well.

The **mouseDown:** method enables an instance of NXColorWell to send its color to another NXColorWell or any other subclass of View that implements the NXDraggingDestination protocol.

See also: NXColorPanel

Instance Variables

NXColor **color**;

color	The current color of the NXColorWell.

Method Types

Initializing an NXColorWell	– initFrame:
Multiple NXColorWells	+ activeWellsTakeColorFrom: + activeWellsTakeColorFrom:continuous: + deactivateAllWells
Drawing	– drawSelf:: – drawWellInside:
Handling events	– acceptsFirstMouse – mouseDown: – setContinuous: – isContinuous
Activating and enabling	– activate: – deactivate – isActive – setEnabled:
Setting color	– setColor: – color – takeColorFrom:
Borders	– setBordered: – isBordered
Target and action	– setTarget: – target – setAction: – action
Archiving	– awake

Class Methods

activeWellsTakeColorFrom:

+ **activeWellsTakeColorFrom:***sender*

This method changes the color of all active NXColorWells by invoking their **takeColorFrom:** method with *sender* as the argument. Returns the NXColorWell class object.

See also: – activate:, + activeWellsTakeColorFrom:continuous:, – deactivate, + deactivateAllWells, – takeColorFrom:

activeWellsTakeColorFrom:continuous:

+ **activeWellsTakeColorFrom:***sender* **continuous:**(BOOL)*flag*

If *flag* is YES, this method changes the color of all active NXColorWells that are continuous; If NO, all active NXColorWells, continuous or not, change their color. NXColorWells are updated by invoking their **takeColorFrom:** method with *sender* as the argument. Use this method in a modal event loop with YES as *flag* if you want active NXColorWells to continuously update to reflect the current color of *sender*. Returns the NXColorWell class object.

See also: – **activate:**, – **deactivate**, + **deactivateAllWells**, – **isContinuous**, – **setContinuous:**, – **takeColorFrom:**

deactivateAllWells

+ **deactivateAllWells**

Deactivates all currently active NXColorWells. Returns the NXColorWell class object.

See also: – **activate:**, – **deactivate**

Instance Methods

acceptsFirstMouse

– (BOOL)**acceptsFirstMouse**

Returns YES. NXColorWells by default accept mouse clicks.

action

– (SEL)**action**

Returns the action sent by the NXColorWell to its target.

activate:

– (int)**activate:**(int)*exclusive*

If *exclusive* is YES, this method activates the receiving NXColorWell and deactivates any other active NXColorWells. If NO, this method activates the receiving NXColorWell and keeps previously active NXColorWells active. Redraws the receiver. An active

NXColorWell will have its color updated when the NXColorPanel's current color changes (continuously, if set to do so).

This method returns the number of active NXColorWells.

See also: + activeWellsTakeColorFrom:, – deactivate, – isContinuous

awake

– awake

Performs additional initialization after the receiver is unarchived. Returns **self**.

color

– (NXColor)**color**

Returns the color of the NXColorWell.

See also: – acceptColor:atPoint, – setColor:, – takeColorFrom:

deactivate

– deactivate

Sets the NXColorWell to inactive and redraws it. Returns **self**.

drawSelf::

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

Draws the entire NXColorWell, including its border. Returns **self**.

drawWellInside:

– drawWellInside:(const NXRect *)*insideRect*

Draws the inside of the NXColorWell only, the area where the color is displayed. Returns **self**.

initFrame:

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

Initializes and returns the receiver, a new instance of NXColorPanel within *theFrame*. By default, the color is NX_COLORWHITE and the NXColorWell is bordered and inactive. Returns **self**.

isActive

– (BOOL)**isActive**

Returns YES if the receiving NXColorWell is active, NO if not active.

See also: – **activate:**

isBordered

– (BOOL)**isBordered**

Returns YES if the receiving NXColorWell is bordered.

See also: – **setBordered:**

isContinuous

– (BOOL)**isContinuous**

Returns YES if the receiving NXColorWell will update its color and send its action message to its target when the class receives the message **activeWellsTakeColorFrom:***sender* **continuous:**YES.

See also: – **setContinuous:**

mouseDown:

– **mouseDown:**(NXEvent *)*theEvent*

Makes the receiver the only active NXColorWell if *theEvent* is on the border of the NXColorWell; begins dragging the NXColorWell's color if *theEvent* is in the colored area of the NXColorWell. When color dragging begins, this method lets the user drag the color from the NXColorWell to another NXColorWell or to another View that implements the NXDraggingDestination protocol. Returns **self**.

You never invoke this method. It's sent when an NX_MOUSEDOWN event occurs within the bounds of the NXColorWell.

See also: – activate, – deactivate, – isActive

setAction:

– **setAction:**(SEL) *aSelector*

Sets the action method of the NXColorWell. The action message is sent to the target by NXColorWell's **takeColorFrom:** method. Returns **self**.

setBordered:

– **setBordered:**(BOOL)*flag*

If flag is YES, sets the NXColorWell to display its border. Redraws the receiver and returns **self**.

setColor:

– **setColor:**(NXColor)*color*

Sets the color of the NXColorWell to *color*. Redraws the receiver and returns **self**.

setContinuous:

– **setContinuous:**(BOOL)*flag*

If *flag* is YES, the NXColorWell will update its color and sends its action message to its target each time the class receives an **activeWellsTakeColorFrom:***sender* **continuous:**YES message. If NO, the NXColorWell won't be updated in response to this message. Returns **self**.

See also: – isContinuous

setEnabled:

– **setEnabled:**(BOOL)*flag*

If *flag* is YES, the receiving NXColorWell is enabled. If NO, the receiver is disabled. An NXColorWell cannot be both disabled and active; enabling an NXColorWell doesn't activate it. Returns **self**.

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

setTarget:

– **setTarget:***anObject*

Sets the target of the NXColorWell to *anObject*. The action message is sent to the target by NXColorWell's **takeColorFrom:** method. Returns **self**.

takeColorFrom:

– **takeColorFrom:***sender*

Causes the receiving NXColorWell to set its color by sending a **color** message to *sender*. Sends the NXColorWell's action message to its target and returns **self**.

See also: – **color**

target

– **target**

Returns the target of the NXColorWell. The action message is sent to the target by NXColorWell's **takeColorFrom:** method. Returns **self**.

See also: – **setTarget:**

NXCursor

Inherits From: Object

Declared In: appkit/NXCursor.h

Class Description

An NXCursor holds an image that can become the image that the Window Server can display for the cursor. A **set** message makes the receiver the current cursor:

```
[myNXCursor set];
```

For automatic cursor management, an NXCursor can be assigned to a cursor rectangle within a Window. When the Window is key and the user moves the cursor into the rectangle, the NXCursor is automatically set to be the current cursor. It ceases to be the current cursor when the cursor leaves the rectangle. The assignment is made using View's **addCursorRect:cursor:** method, usually inside a **resetCursorRects** method:

```
- resetCursorRects
{
    [self addCursorRect:&someRect cursor:theNXCursorObject];
    return self;
}
```

This is the recommended way of associating a cursor with a particular region inside a window. However, the NXCursor class provides two other ways of setting the cursor:

- The class maintains its own stack of cursors. Pushing an NXCursor instance on the stack sets it to be the current cursor. Popping an NXCursor from the stack sets the next NXCursor in line, the one that's then at the top of the stack, to be the current cursor.
- An NXCursor can be made the owner of a tracking rectangle and told to set itself when it receives a mouse-entered or mouse-exited event.

The Application Kit provides two ready-made NXCursor instances and assigns them to global variables:

NXArrow	The standard arrow cursor
NXIBeam	The cursor that's displayed over editable or selectable text

There's no NXCursor instance for the wait cursor. The wait cursor is displayed automatically by the system, without any required program intervention.

Instance Variables

```
NXPoint hotSpot;
struct _csrFlags {
    unsigned int  onMouseExited:1;
    unsigned int  onMouseEntered:1;
} cFlags;
id  image;
```

hotSpot	The point in the cursor image whose location on the screen is reported as the cursor's location.
cFlags.onMouseExited	A flag indicating whether to set the cursor when the NXCursor object receives a mouse-exited event.
cFlags.onMouseEntered	A flag indicating whether to set the cursor when the NXCursor object receives a mouse-entered event.
image	The cursor image, an NXImage object.

Method Types

Initializing a new NXCursor object	– init – initFromImage:
Defining the cursor	– setImage: – image – setHotSpot:
Setting the cursor	– push – pop + pop – set – setOnMouseEntered: – setOnMouseExited: – mouseEntered: – mouseExited: + currentCursor
Archiving	– read: – write:

Class Methods

currentCursor

+ currentCursor

Returns the last NXCursor that received a **set** message.

See also: **– set**, **– push**, **+ pop**, **– mouseEntered:**, **– mouseExited:**,

pop

+ pop

Removes the NXCursor at the top of the cursor stack, and sets the NXCursor that was beneath it to be the current cursor. Returns **self** (the class object).

This method can be used in conjunction with the **push** method to manage a group of cursors within a local context. Every **push** should be balanced by a subsequent **pop**. When the last remaining cursor is popped from the stack, the Application Kit restores a cursor appropriate for the larger context.

The **pop** instance method provides the same functionality as this class method.

See also: **– push**

Instance Methods

image

– image

Returns the NXImage object that supplies the cursor image for the receiving NXCursor, or **nil** if no image has been set.

See also: **– initFromImage:**, **– setImage:**

init

– init

Initializes the receiver, a newly allocated NXCursor instance, by sending it an **initFromImage:** message with **nil** as the argument. This doesn't assign an image to the

object. An image must then be set (with the **setImage:** method) before the cursor can be used. Returns **self**.

See also: – **setImage:**, – **initFromImage:**

initFromImage:

– **initFromImage:***image*

Initializes the receiver, a newly allocated NXCursor instance, by setting the image it will use to *image*, an NXImage object. The image must be at least 16 pixels wide by 16 pixels high. If the image is smaller than 16-by-16, an error is generated when the application tries to use the cursor, and the previous cursor remains in use. If the image is larger than 16-by-16, only the lower-left 16-by-16 pixels of the image will be displayed. The default hot spot is at the upper left corner of the 16-by-16 square.

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

See also: – **setHotSpot:**, – **setImage:**

mouseEntered:

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

Responds to a mouse-entered event by setting the receiver to be the current cursor, but only if enabled to do so by a previous **setOnMouseEntered:** message. This method does not push the receiver on the cursor stack. Returns **self**.

See also: – **setOnMouseEntered:**

mouseExited:

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

Responds to a mouse-exited event by setting the receiver to be the current cursor, but only if enabled to do so by a previous **setOnMouseExited:** message. This method does not push the receiver on the cursor stack. Returns **self**.

See also: – **setOnMouseExited:**

pop

– **pop**

Removes the topmost NXCursor object, not necessarily the receiver, from the cursor stack, and makes the next NXCursor down the current cursor. Returns **self**.

This method is a cover for the class method of the same name.

See also: + **pop**, – **push**

push

– **push**

Puts the receiving NXCursor on the cursor stack and sets it to be the Window Server's cursor. Returns **self**.

This method can be used in conjunction with the **pop** method to manage a group of cursors within a local context. Every **push** should be matched by a subsequent **pop**.

See also: + **pop**

read:

– **read:**(NXTypedStream *)*stream*

Writes the NXCursor, including the image, to *stream*. Returns **self**.

See also: – **write:**

set

– **set**

Makes the NXCursor the cursor displayed by the Window Server, and returns **self**. This method doesn't push the receiver on the cursor stack.

setHotSpot:

– **setHotSpot:**(const NXPoint *)*aPoint*

Sets the point on the cursor that will be used to report its location. The point is specified relative to a flipped coordinate system with an origin at the upper left corner of the cursor image and coordinate units equal to those of the base coordinate system. The point should

not have any fractional coordinates, meaning that it should lie at the corner of four pixels. The point selects the pixel below it and to its right. This pixel is the one part of the cursor image that's guaranteed never to be off-screen.

When the pixel selected by the hot spot lies inside a rectangle (say a button), the cursor is said to be over the rectangle. When the pixel is outside the rectangle, the cursor is taken to be outside the rectangle, even if other parts of the image are inside.

The default hot spot is at the upper left corner of the image—(0,0) in its flipped coordinate system. Returns **self**.

setImage:

– **setImage:***image*

Assigns a new cursor *image* to the receiving NXCursor, and returns **self**. *image* should be an NXImage object for an image that's 16 pixels wide by 16 pixels high. If the image is smaller than 16-by-16, an error is generated when the application tries to use the cursor, and the previous cursor remains in use. If the image is larger than 16-by-16, only the lower-left 16-by-16 pixels of the image will be displayed.

Resetting the image of an NXCursor while it is the current cursor may have unpredictable results.

See also: – **image**, – **initFromImage:**

setOnMouseEntered:

– **setOnMouseEntered:**(BOOL)*flag*

Determines whether the NXCursor should set itself to be the current cursor when it receives a **mouseEntered:** event message. To be able to receive the event message, an NXCursor must first be made the owner of a tracking rectangle by Window's **setTrackingRect:inside:owner:tag:left:right:** method.

Cursor rectangles are a more convenient way of associating cursors with particular areas within a window.

Returns **self**.

See also: – **mouseEntered:**, – **setTrackingRect:inside:owner:tag:left:right:** (Window)

setOnMouseExited:

– **setOnMouseExited:**(BOOL)*flag*

Determines whether the NXCursor should set itself to be the current cursor when it receives a **mouseExited:** event message. To be able to receive the event message, an NXCursor must first be made the owner of a tracking rectangle by Window's **setTrackingRect:inside:owner:tag:left:right:** method.

Cursor rectangles are a more convenient way of associating cursors with particular areas within windows.

Returns **self**.

See also: – **mouseExited:**, – **setTrackingRect:inside:owner:tag:left:right:** (Window)

write:

– **write:**(NXTypedStream *)*stream*

Writes the NXCursor and its image to *stream*. Returns **self**.

See also: – **read:**

NXCustomImageRep

Inherits From: NXImageRep : Object

Declared In: appkit/NXCustomImageRep.h

Class Description

An NXCustomImageRep is an object that uses a delegated method to render an image. When called upon to produce the image, it sends a message to have the method performed.

Like most other kinds of NXImageReps, an NXCustomImageRep is generally used indirectly, through an NXImage object. To be useful to the NXImage, it must be able to provide some information about the image. The following methods, inherited from the NXImageRep class, inform the NXCustomImageRep about the size of the image, whether it can be drawn in color, and so on. Use them to complete the initialization of the object.

setSize:
setNumColors:
setAlpha:
setPixelsHigh:
setPixelsWide:
setBitsPerSample:

Instance Variables

SEL **drawMethod**;
id **drawObject**;

drawMethod	The method that draws the image.
drawObject	The object that receives messages to perform the **drawMethod**.

Method Types

Initializing a new NXCustomImageRep	– initDrawMethod:inObject:
Drawing the image	– draw
Archiving	– read: – write:

Instance Methods

draw

– (BOOL)**draw**

Sends a message to have the image drawn. Returns YES if the message is successfully sent, and NO if not. The message will not be sent if the intended receiver is **nil** or it can't respond to the message.

See also: – **drawAt:** (NXImageRep), – **drawIn:** (NXImageRep)

init

Generates an error message. This method cannot be used to initialize an NXCustomImageRep. Use **initDrawMethod:inObject:** instead.

See also: – **initDrawMethod:inObject:**

initDrawMethod:inObject:

– **initDrawMethod:**(SEL)*aSelector* **inObject:***anObject*

Initializes the receiver, a newly allocated NXCustomImageRep instance, so that it delegates responsibility for rendering the image to *anObject*. When the NXCustomImageRep receives a **draw** message, it will in turn send a message to *anObject* to perform the *aSelector* method. The *aSelector* method should take only one argument, the **id** of the NXCustomImageRep. It should draw the image at location (0.0, 0.0) in the current coordinate system.

Returns **self**.

read:

– **read:**(NXTypedStream *)*stream*

Reads the NXCustomImageRep from the typed stream *stream*.

See also: – **write:**

write:

– **write:**(NXTypedStream *)*stream*

Writes the NXCustomImageRep to the typed stream *stream*. The object that's delegated to draw the image is not explicitly written.

See also: – **read:**

NXDataLink

Inherits From: Object

Declared In: appkit/NXDataLink.h

Class Description

An NXDataLink defines a single data link between a selection in a source document and a dependent, dynamically updated selection in a destination document.

A data link is typically created when linkable data is copied to the pasteboard. First, an NXSelection object describing the data is created. Then a link to that selection is created using **initLinkedToSourceSelection:managedBy:supportingTypes:count:**. The link can then be written to the pasteboard using **writeToPasteboard:**. Usually, after the link has been written to the pasteboard (or saved to a file using **writeToFile:**) the link is freed because it is generally of no further use to the source application.

Once the data and link have been written to the pasteboard, they can be added to a destination document using the Paste and Link command. The implementation responding to this command will paste the data as usual (with the possible exception that the linked data will be pasted in a "display" format rather than a richer, editable format, since the data is to be dependent on the source data rather than fundamentally editable). The destination application will then read the link from the pasteboard using **initFromPasteboard:**, create an NXSelection describing the linked data within the destination document, and will add the link by sending **addLink:at:** to the document's link manager.

When the link is added to the destination document's link manager, it becomes a *destination link*. At this time, the data links implementation establishes a connection with the source document's link manager, which automatically creates a *source link* in the source application; the source link refers to the source selection. Only applications that must update destination links continuously need to be aware of when source links are created; these applications can return YES in response to a **dataLinkManagerTracksLinksIndividually:** message, and then respond to **dataLinkManager:startTrackingLink:** messages to receive notifications that source links are created.

A link that isn't managed by a link manager is a *broken link*. (Both source and destination links have link managers.) All links are broken links when they are created. Links can be explicitly broken (ensuring that they cause no updates) using the **break** method. Broken links (that aren't former source links) can be hooked up as destination links with the **addLink:at:** method. The disposition of a link (destination, source, or broken) can be retrieved with the **disposition** method. Most of the messages defined by the NXDataLink class can be sent to a link of any disposition, but some only make sense when sent to a link with a specific disposition; these are so noted in their method descriptions.

Links of all dispositions (except links to files) maintain an NXSelection object referring to the link's selection in the source document; this selection is returned by the **sourceSelection** method. Links directly to files represent entire files rather than selections in a document; these links are created with **initLinkedToFile:** and have no source selection.

Source and destination links also maintain an NXSelection describing how the data is presented in the destination document; this selection is returned by the **destinationSelection** method.

Instance Variables

None declared in this class.

Method Types

Initializing a link	– initFromFile: – initFromPasteboard: – initLinkedToFile: – initLinkedToSourceSelection:managedBy: supportingTypes:count: – copyFromZone:
Exporting a link	– writeToPasteboard: – saveLinkIn: – writeToFile:
Information about the link	– manager – disposition – linkNumber

Information about the link's source
- sourceAppName
- sourceFilename
- sourceSelection
- openSource
- lastUpdateTime
- types

Information about the link's destination
- destinationAppName
- destinationFilename
- destinationSelection

Updating the link's data
- sourceEdited
- updateDestination
- setUpdateMode:
- updateMode
- break

Instance Methods

break

– break

Breaks the link so the data referred to by its selection will not get updated. The link is removed from its link manager and its destination selection is freed. The link itself is not freed; if it wasn't formerly a source link it can be hooked back up again using NXDataLinkManager's **addLink:at:** method. Alternatively, it can be explicitly freed by the application if the application directly sent the **break** message, or freed by Applications's **delayedFree:** method on receiving a **dataLinkManager:didBreakLink:** notification that the link was broken. (This could happen in response to user input from the data link panel.) Returns **self** if sent to a destination link, does nothing and returns **nil** if sent to a broken link If sent to a source link, the message is forwarded to the destination link; it then returns **self** if the link is successfully broken and **nil** otherwise.

copyFromZone:

– **copyFromZone:**(NXZone *)*zone*

Returns a copy of the receiving data link allocated from *zone*. The copy is essentially linked to the source data, but not hooked up to the destination document. The copy has a copy of the receiver's source selection, has no destination selection, and its disposition is NX_LinkBroken.

See also: – **addLink:at:** (NXDataLinkManager)

destinationAppName

– (const char *)**destinationAppName**

Returns the name (as returned by Application's **appName** method) of the application containing the destination link.

See also: – **sourceAppName**

destinationFilename

– (const char *)**destinationFilename**

Returns the file name of the destination document, as set by any of several NXDataLinkManager methods for the destination document.

See also: – **sourceFilename**

destinationSelection

– (NXSelection *)**destinationSelection**

Returns the destination selection, which describes how the linked data is represented in the destination document.

See also: – **sourceSelection**

disposition

– (NXDataLinkDisposition)**disposition**

Identifies the link as a source link, a destination link, or a broken link by returning one of the following values:

NX_LinkInDestination
NX_LinkInSource
NX_LinkBroken

See also: – **addLink:at:** (NXDataLinkManager),
– **dataLinkManager:startTrackingLink:** (NXDataLinkManager delegate), – **break**

initFromFile:

– **initFromFile:**(const char *)*filename*

Initializes a newly allocated NXDataLink instance from *filename*, a link that was previously saved using the **saveLinkIn:** or **writeToFile:** method. The new link is generally used by adding it to a destination document's link manager with **addLink:at:**. Returns the link if it is successfully initialized; otherwise frees the link and returns **nil**.

See also: – **saveLinkIn:**, – **writeToFile:**, – **addLink:at:** (NXDataLinkManager)

initFromPasteboard:

– **initFromPasteboard:**(Pasteboard *)*pasteboard*

Initializes a newly allocated NXDataLink instance from the pasteboard *pasteboard*. The new link is generally used by adding it to a destination document's link manager with **addLink:at:**.

In order for this method to succeed, a link must have been placed on the pasteboard using **writeToPasteboard:**, or the file name of a saved link (data of type NXFilenamePboardType with an extension of NXDataLinkFilenameExtension) must be on the pasteboard.

Returns the link if it is successfully initialized; otherwise frees the link and returns **nil**.

See also: – **writeToPasteboard:**, – **saveLinkIn:**, – **addLink:at:** (NXDataLinkManager)

initLinkedToFile:

– **initLinkedToFile:**(const char *)*filename*

Initializes a newly allocated NXDataLink instance corresponding to the entire file *filename*. The link is identified as a link to a file because it has no source selection. Returns the link if it is successfully initialized; otherwise frees the link and returns **nil**.

See also: – **addLink:at:** (NXDataLinkManager), – **writeToPasteboard:**, – **sourceSelection**

initLinkedToSourceSelection:managedBy:supportingTypes:count:

– **initLinkedToSourceSelection:**(NXSelection *)*selection*
managedBy:*linkManager*
supportingTypes:(const char *const *)*newTypes*
count:(int)*numTypes*

Initializes a newly allocated NXDataLink instance corresponding to a selection in the source document described by *selection*. *linkManager* is the source document's link manager. *newTypes* is an array (with size *numTypes*) of pointers to the pasteboard types that *linkManager*'s delegate is willing to provide (by **copyToPasteboard:at:cheapCopyAllowed:**) when a user of the link requests the data described by *selection*.

Typically, when the user uses the Copy command to copy linkable data, this method should be invoked to create a link corresponding to the data. The new link should be added to the pasteboard (by **writeToPasteboard:**) and immediately freed, since it will usually be of no further use to the source document. Many links so placed on the pasteboard will go unused and will simply be discarded when the pasteboard changes. If state identifying *selection* must be saved in the source document, the link manager's delegate should find out whether the selection is used by implementing **dataLinkManager:startTrackingLink:** (to discover if the selection gets used) and the Pasteboard owner's method **pasteboardChangedOwner:** (to discover when the pasteboard has changed, precluding the use of a link that was placed there).

Returns the new link.

See also: – **copyToPasteboard:at:cheapCopyAllowed:** (NXDataLinkManager delegate), – **dataLinkManager:startTrackingLink:** (NXDataLinkManager delegate), – **pasteboardChangedOwner:** (Pasteboard owner)

lastUpdateTime

– (time_t)**lastUpdateTime**

Returns the last time the link was updated. A link could be updated for many reasons; for example, a message could be sent to the source document's link manager telling it that its document was saved, or the link could be brought up to date with an **updateDestination** message.

See also: – **setLinksVerifiedByDelegate:** (NXDataLinkManager), – **documentSaved** (NXDataLinkManager)

linkNumber

– (NXDataLinkNumber)**linkNumber**

Returns a destination link's link-number, which may be useful in identifying the link. This number is constant through the life of the document, and unique among the document's links; it is not meaningful in source links.

manager

– (NXDataLinkManager *)**manager**

Returns the link's link manager, or **nil** if it doesn't have one. (For example, returns **nil** if the link is broken.)

openSource

– **openSource**

Opens the document corresponding to the link's source selection. This message only has meaning when sent to a destination link. Returns **self** if the source document is successfully opened, **nil** otherwise.

See also: – **app:openFile:type:** (Application delegate), – **showSelection:** (NXDataLinkManager delegate)

saveLinkIn:

– **saveLinkIn:**(const char *)*directoryName*

Saves the link with a file name provided by the user. This method should be invoked through the Publish Selection command. It runs the SavePanel to prompt the user for a filename to save the link to. The SavePanel's initial directory is provided in *directoryName*. It then writes the link using the **writeToFile:** method. Returns **self** if the link is successfully saved; **nil** otherwise.

See also: – **initFromFile:**

setUpdateMode:

– **setUpdateMode:**(NXDataLinkUpdateMode)*mode*

Sets the link's update mode to *mode*, which must be one of the following values:

NX_UpdateContinuously
NX_UpdateWhenSourceSaved
NX_UpdateManually
NX_UpdateNever

A mode of NX_UpdateContinuously updates the link's destination data every time a **sourceEdited** message is sent to the source link. A mode of NX_UpdateWhenSourceSaved updates the link's destination data every time a **documentSaved** (or related) message is sent to the source link's link manager. A mode of NX_UpdateManually updates the link's destination data every time a **updateDestination** message is sent to the destination link; this message can be sent programmatically or by the data link panel. A mode of NX_UpdateNever makes the link never update; once a destination link has been set to this mode, it can't be set back to any other mode until it is broken. (This mode is used for link buttons, for example.)

This message only has meaning when sent to a destination link or a broken link. Returns **self**.

See also: – **updateMode**, – **break**

sourceAppName

– (const char *)**sourceAppName**

Returns the name (as returned by Application's **appName** method) of the application containing the source link.

See also: – **destinationAppName**

sourceEdited

– **sourceEdited**

Sent to a source link to inform it that the data referred to by its source selection has changed. If the link's destination link has been set to update continuously, the destination will be updated.

This message only has meaning if sent to a source link. An application will only know of the source links that are being used in its document if the data link manager's delegate tracks links individually and responds to **dataLinkManager:startTrackingLink:** messages. However, an application doesn't need to track source links individually unless it wishes to allow continuous updating.

Returns **self** unless the link's destination is set to continuously update and the update fails; in this case, the method returns **nil**.

See also: – **dataLinkManager:startTrackingLink:** (NXDataLinkManager delegate)

sourceFilename

– (const char *)**sourceFilename**

Returns the file name of the source document, as set by any of several NXDataLinkManager methods for the source document.

See also: – **destinationFilename**

sourceSelection

– (NXSelection *)**sourceSelection**

Returns the source selection, or **nil** if the link refers to an entire file (in which case the source file can be retrieved from **sourceFilename**).

See also: – **destinationSelection**

types

– (const NXAtom *)**types**

Returns the pasteboard types that the source document can provide when the data for the link's source selection is required.

See also: – **copyToPasteboard:at:cheapCopyAllowed:** (NXDataLinkManager delegate)

updateDestination

– **updateDestination**

Updates the data referred to by the link's destination selection. This message can be sent to a source link or a destination link. If it's sent to a destination link, it will usually open the source document if it isn't already open. If it is sent to a source link, it will usually force the destination data to be immediately updated (which is generally less desirable than sending the source link a **sourceEdited** message, since that would allow the update to occur at the normal time). If the destination must be updated, it will be done by sending a **pasteFromPasteboard:at:** or **importFile:at:** message to the destination link manager's delegate.

See also: – **pasteFromPasteboard:at:** (NXDataLinkManager delegate)

updateMode

– (NXDataLinkUpdateMode)**updateMode**

Returns the link's update mode, which determines when the data referred to by the link's destination selection will be updated. Possible return values are:

NX_UpdateContinuously
NX_UpdateWhenSourceSaved
NX_UpdateManually
NX_UpdateNever

A description of these values can be found in the method description for **setUpdateMode:**.

writeToFile:

– **writeToFile:**(const char *)*filename*

Writes the link into the file *filename*. This allows selections to be published by the file system; the link can be read in later using **initFromFile:**. Returns **self** if the link is successfully written, **nil** otherwise.

See also: – **initLinkedToSourceSelection:managedBy:supportingTypes:count:**, – **writeToPasteboard:**, – **saveLinkIn:**, – **initFromFile:**

writeToPasteboard:

– **writeToPasteboard:**(Pasteboard *)*pasteboard*

Writes the link onto the pasteboard *pasteboard*, allowing other applications to paste both copied data and the link referring to that data. When a link is written to a pasteboard, the type **NXDataLinkPboardType** must be included in the pasteboard's types, either using **declareTypes:num:owner:** or **addTypes:num:owner:**. The link can be read in later using **initFromPasteboard:**. Returns **self** if the link is successfully written, **nil** otherwise.

See also: – **initLinkedToSourceSelection:managedBy:supportingTypes:count:**, – **writeToFile**, – **initFromPasteboard:**

NXDataLinkManager

Inherits From: Object

Conforms To: NXSenderIsInvalid

Declared In: appkit/NXDataLinkManager.h

Class Description

An NXDataLinkManager manages data linked from and into a document. If an application supports data linking, a data link manager should be instantiated for every document the application creates. A data link manager must be assigned a delegate that assists it in keeping the document up-to-date; this delegate must implement some or all of the methods listed in the "Methods Implemented by the Delegate" section of this class specification. In addition, the delegate must keep the link manager informed of the state of the document, sending it messages whenever the document is edited, saved, or otherwise altered.

Instance Variables

None declared in this class.

Method Types

Initializing and freeing a link manager
- initWithDelegate:
- initWithDelegate:fromFile:
- free

Adding and removing links
- addLink:at:
- addLinkAsMarker:at:
- writeLinksToPasteboard:
- addLinkPreviouslyAt:fromPasteboard:at:
- breakAllLinks

Informing the link manager of document status

- documentClosed
- documentEdited
- documentReverted
- documentSaved
- documentSavedAs:
- documentSavedTo:

Getting and setting information about the link manager

- filename
- isEdited
- setLinksVerifiedByDelegate:
- areLinksVerifiedByDelegate
- delegate
- setInteractsWithUser:
- interactsWithUser

Getting and setting information about the manager's links

- setLinkOutlinesVisible:
- areLinkOutlinesVisible
- findDestinationLinkWithSelection:
- prepareEnumerationState:forLinksOfType:
- nextLinkUsing:

Instance Methods

addLink:at:

– **addLink:**(NXDataLink *)*link* **at:**(NXSelection *)*selection*

Adds the link *link* to the document, indicating that the data in the document described by *selection* is dependent upon the link. This method is invoked as part of the Paste and Link command to actually link in the data that was just pasted. It can also be used at other times; for example, to link to files that are dragged into the document.

This method makes *link* a destination link and sets *selection* as *link*'s destination selection. When the link's source is modified, the link manager's delegate will be sent a **pasteFromPasteboard:at:** or **importFile:at:** message with *selection* as an argument, indicating that the destination data must be updated.

Returns **self** if the link is successfully added, **nil** if it isn't. There are several situations that will result in failure to add the link, such as an inability to resolve the link to its source, so it's important to check the return value of this method and undo the requested operation if the linking fails.

addLinkAsMarker:at:

– **addLinkAsMarker:**(NXDataLink *)*link* **at:**(NXSelection *)*selection*

Incorporates *link* into the document as a marker. This method is used to implement link buttons that allow access to the link's source, but are never asked to receive data from the source document. The link button in the destination document is described by *selection*. This method adds the link and, upon success, sets its the link's update mode to NX_UpdateNever. Returns **self** upon success, **nil** otherwise.

The named images "NXLinkButton" and "NXLinkButtonH" can be used (through NXImage's **findImageNamed:** method) to represent ordinary and highlighted link buttons, respectively. These images are shared, so you must copy them (using NXImage's **copy** method) if you need to scale them to a different size.

See also: – **addLink:at:**

addLinkPreviouslyAt:fromPasteboard:at:

– (NXDataLink *)**addLinkPreviouslyAt:**(NXSelection *)*oldSelection*
fromPasteboard:(Pasteboard *)*pasteboard*
at:(NXSelection *)*selection*

Creates and adds a new destination link corresponding to the same source data as the link described by the destination selection *oldSelection*. The new link's destination selection is provided in *selection*. This method is useful if you paste data that is already linked. It's similar to copying the old link and adding it using **addLink:at:**, except you specify the old destination selection rather than the old link. Before invoking this method, the document's links must be written to the pasteboard *pasteboard* using **writeLinksToPasteboard:**. Returns the new link if it's successfully added, or **nil** if the link can't be added or no link for *oldSelection* existed.

areLinkOutlinesVisible

– (BOOL)**areLinkOutlinesVisible**

Used to inform the link manager's delegate of whether link outlines should be drawn around linked destination data. When the delegate receives a **dataLinkManagerRedrawLinkOutlines:** message, it should query the link manager with an **areLinkOutlinesVisible** message. If this message returns YES, the delegate should call the **NXFrameLinkRect()** function to draw a distinctive link outline around the dependent data.

See also: – **setLinkOutlinesVisible**:

areLinksVerifiedByDelegate

– (BOOL)**areLinksVerifiedByDelegate**

Return YES if the link manager's delegate will be asked to verify whether data based on the delegate's source links needs to be updated. If so, the delegate should implement the **dataLinkManager:isUpdateNeededForLink:** method. Returns NO by default, but the application can change this by sending the link manager a **setLinksVerifiedByDelegate:** message.

breakAllLinks

– **breakAllLinks**

Breaks all the destination links in the document by sending each link a **break** message. This method is typically invoked by the application's data link panel in response to user input. Returns **self**.

See also: – **break** (NXDataLink), – **pickedBreakAllLinks:** (NXDataLinkPanel)

delegate

– **delegate**

Returns the data link manager's delegate, the object that will be sent messages to provide source data, paste destination data, and help the data link manager keep links up-to-date.

See also: – **initWithDelegate:**

documentClosed

– **documentClosed**

An application should send this message to the link manager to inform it that the manager's document has been closed. Returns **self**.

documentEdited

– **documentEdited**

An application should send this message to the link manager to inform it that the manager's document has been edited. If the delegate doesn't track source links individually, this

method marks all source links as dirty, indicating that the dependant destination data will eventually need to be updated. Returns **self**.

See also: – **dataLinkManagerTracksLinksIndividually:** (NXDataLinkManager delegate)

documentReverted

– **documentReverted**

An application should send this message to the link manager to inform it that the manager's document has been reverted to the last saved copy. This method then restores the link manager and its links to their last saved state (from the last time the link manager received a **documentSaved** or **documentSavedAs:** message). Returns **self**.

documentSaved

– **documentSaved**

An application should send this message to the link manager to inform it that the manager's document has been saved. This method stores the document's destination links and, if necessary, initiates updates of other documents dependent upon the document's source links. Returns **self**.

documentSavedAs:

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

An application should send this message to the link manager to inform it that the manager's document has been saved as the file specified by the full pathname *path*. This method stores the document's destination links. It also discards the manager's source links, since the documents for those links are not dependent upon the newly saved file. Returns **self**.

documentSavedTo:

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

An application should send this message to the link manager to inform it that a copy of the manager's document has been saved to the file specified by the full pathname *path*. This method stores the appropriate link information along with the file, and returns **self**.

filename

– (const char *)**filename**

Returns the name of the file for the link manager's document. This is the name that was set with the **initWithDelegate:fromFile:** or **documentSavedAs:** method.

findDestinationLinkWithSelection:

– (NXDataLink *)**findDestinationLinkWithSelection:**(NXSelection *)*destSel*

Returns the destination link for the selection *destSel*, or **nil** if the document has no link for that selection. This method will tell you if the given selection is linked to a source.

free

– **free**

Notifies the link managers of dependent documents that the link manager is going away, and frees the objects and storage held by the link manager.

init

– **init**

There is no need to call this method; use one of the other **init...** methods to initialize a newly allocated NXDataLinkManager instance for a new document.

See also: – **initWithDelegate:fromFile:**

initWithDelegate:

– **initWithDelegate:***anObject*

Initializes and returns a newly allocated NXDataLinkManager instance for a new document. The link manager's delegate, specified by *anObject*, will be expected to provide source data, paste destination data, and help the data link manager keep links up-to-date. Before data in the document can be linked to, the document will have to be saved and the link manager will have to be informed of the document's name by a **documentSavedAs:** message.

See also: – **initWithDelegate:fromFile:**

initWithDelegate:fromFile:

– **initWithDelegate:***anObject* **fromFile:**(const char *)*path*

Initializes a newly allocated NXDataLinkManager instance for a new document. The link manager's delegate, specified by *anObject*, will be expected to to provide source data, paste destination data, and help the data link manager keep links up-to-date. The document's file is specified by the full path *path*. The file must exist or initialization will fail.

See "Methods Implemented by the Delegate" at the end of this class specification for information about the methods the delegate should implement to assist the link manager.

Returns the new link manager upon success; frees the allocated storage and returns **nil** if initialization fails.

See also: – **initWithDelegate:fromFile:**

interactsWithUser

– (BOOL)**interactsWithUser**

Returns YES if the link manager should display alert panels when problems with links occur, NO if the displays are to be suppressed. This value is set with the **setInteractsWithUser:** method; the default value is YES.

isEdited

– (BOOL)**isEdited**

Returns YES if the document has been edited since the last save, or NO if the file for the document is up-to-date. The document's edited state is set by the **documentEdited** method, and cleared by **documentSaved** and related methods.

nextLinkUsing:

– (NXDataLink *)**nextLinkUsing:**(NXLinkEnumerationState *)*state*

Returns the link manager's next link based on *state*. *state* must be initially prepared using **prepareEnumerationState:forLinksOfType:** to allow the caller to retrieve each of the application's destination or source links in turn. This method return **nil** if there are no more links.

prepareEnumerationState:forLinksOfType:

– **prepareEnumerationState:**(NXLinkEnumerationState *)*state*
forLinksOfType:(NXDataLinkDisposition)*srcOrDest*

Prepares the variable indicated by *state* to allow the application to retrieve each of the link manager's links, one at a time, with later invocations of **nextLinkUsing:**. *srcOrDest* must be either NX_LinkInDestination or NX_LinkInSource to indicate whether the **nextLinkUsing:** method is to return the next destination link or the next source link, respectively. Returns **self** if there is one or more links of the requested type, or **nil** if there is none.

setInteractsWithUser:

– **setInteractsWithUser:**(BOOL)*flag*

Instructs the link manager as to whether it should display alert panels when problems with links occur. If *flag* is YES (the default value), alert panels will be displayed. Returns **self**.

See also: – **interactsWithUser**

setLinkOutlinesVisible:

– **setLinkOutlinesVisible:**(BOOL)*flag*

Sets the internal flag indicating to the link manager's delegate whether link outlines ought to be displayed; this value can be returned by **areLinkOutlinesVisible**.

If the link manager's delegate implements the **dataLinkManagerRedrawLinkOutlines:** method, this message will be sent to the delegate and it should either display link outlines using **NXFrameLinkRect()** or erase link outlines if they were previously displayed, based on the return value of **areLinkOutlinesVisible**.

Returns **self**.

setLinksVerifiedByDelegate:

– **setLinksVerifiedByDelegate:**(BOOL)*flag*

Sets whether the update status of links will be individually verified by the link manager's delegate. If flag is YES, the delegate must implement the **dataLinkManager:isUpdateNeededForLink:** method to tell the link manager if data based on a source link needs to be updated.

By default, the update status of an individual link isn't verified by the delegate, so the link manager verifies a link based on its last update time. An example where this verification could be incorrect might be a link to a database query; if the query itself doesn't change, the link manager might return that data is up-to-date, even though the database referred to by the query might have changed.

See also: – **areLinksVerifiedByDelegate**

writeLinksToPasteboard:

– **writeLinksToPasteboard:**(Pasteboard *)*pasteboard*

Writes all the link manager's links to the pasteboard *pasteboard* in preparation for an invocation of **addLinkPreviouslyAt:fromPasteboard:at:**, which will expect to find one link matching its specified selection.

The links are written with Pasteboard's **addTypes:num:owner:** method, which doesn't change the pasteboard's owner or change count, using a private pasteboard type.

Methods Implemented by the Delegate

copyToPasteboard:at:cheapCopyAllowed:

– **copyToPasteboard:**(Pasteboard *)*pasteboard*
at:(NXSelection *)*selection*
cheapCopyAllowed:(BOOL)*flag*

Implemented by the link manager's delegate to supply the source data described by *selection* on the pasteboard *pasteboard*. *selection* was previously provided by the application when it created an NXDataLink using **initLinkedToSourceSelection:managedBy:**.

Since the Pasteboard works lazily, the delegate doesn't have to provide all data representations at this time; it simply has to declare the pasteboard types it's willing to provide for *selection*. Normally, the delegate must put at least one representation on the pasteboard in order to generate any of the specified types when one is requested. However, if *flag* is YES, the system guarantees that no events will be processed by the application before the delegate is requested to provide the specified data; in this case, the application doesn't necessarily have to write any data representations to the pasteboard. This method should return **self** upon success, or **nil** if the selection can't be resolved.

See also: – **pasteFromPasteboard:at:** (NXDataLinkManager delegate), – **declareTypes:num:owner:** (Pasteboard), – **pasteboard:provideData:** (Pasteboard owner)

createSelection

– (NXSelection *)**createSelection**

Never invoked by the system.

dataLinkManager:didBreakLink:

– **dataLinkManager:**(NXDataLinkManager *)*sender*
didBreakLink:(NXDataLink *)*link*

If this method is implemented by the delegate, it will be invoked to inform the delegate that the destination link *link* was broken and thus data based on *link*'s destination selection will no longer be updated.

The link shouldn't be be sent a **free** message at this time, because the method that invoked **dataLinkManager:didBreakLink:** may still reference the link. However, the link can be freed with Application's **delayedFree:** method. Alternatively, the link could be kept around for a while in order to allow the break operation to be undone; if this is requested, the link could be re-added with **addLink:at:**.

See also: – **break** (NXDataLink), – **destinationSelection** (NXDataLink)

dataLinkManager:isUpdateNeededForLink:

– (BOOL)**dataLinkManager:**(NXDataLinkManager *)*sender*
isUpdateNeededForLink:(NXDataLink *)*link*

A delegate that sends a **setLinksVerifiedByDelegate:** message to the link manager (indicating that the update status for individual links will be verified by the delegate) should implement this method and return YES if the source data identified by link's source selection has been modified since the link's last update time.

See also: – **lastUpdateTime** (NXDataLink)

dataLinkManager:startTrackingLink:

– **dataLinkManager:**(NXDataLinkManager *)*sender*
startTrackingLink:(NXDataLink *)*link*

Informs the delegate that another document has established a data link to the link manager's document. The delegate need only implement this method if it returns YES in response to

a **dataLinkManagerTracksLinksIndividually:** message. *link* is a newly added source link; the data that it applies to is identified by *link*'s source selection.

See also: – **dataLinkManagerTracksLinksIndividually:**

dataLinkManager:stopTrackingLink:

– **dataLinkManager:**(NXDataLinkManager *)*sender*
stopTrackingLink:(NXDataLink *)*link*

Informs the delegate that the former source link *link* is no longer linked to the document. There are many reasons that a link might be removed; the destination document could get closed, the link could be explicitly broken, or the destination application might have died.

See also: – **dataLinkManagerTracksLinksIndividually:**

dataLinkManagerCloseDocument:

– **dataLinkManagerCloseDocument:**(NXDataLinkManager *)*sender*

Never invoked by the system.

dataLinkManagerDidEditLinks:

– **dataLinkManagerDidEditLinks:**(NXDataLinkManager *)*sender*

Informs the delegate that link data has been modified. Since the link data is stored alongside the document's data and should be considered part of the document, the delegate should use this notification to mark the document as edited.

dataLinkManagerRedrawLinkOutlines:

– **dataLinkManagerRedrawLinkOutlines:**(NXDataLinkManager *)*sender*

If the delegate implements this method, it will be invoked any time the manager is instructed to show or hide link outlines through **setLinkOutlinesVisible:**. This method should query the link manager with **areLinkOutlinesVisible** to find out if link outlines should be displayed. If so, it should invoke **NXFrameLinkRect()** to draw a distinctive outline around the linked data; otherwise it should display the data without outlines.

dataLinkManagerTracksLinksIndividually:

– (BOOL)**dataLinkManagerTracksLinksIndividually:**
(NXDataLinkManager *)*sender*

If the delegate implements this method it should return whether it's willing to track links individually. If the delegate doesn't implement this method, links are not individually tracked. If the delegate implements this method and returns YES, it should also implement **dataLinkManager:startTrackingLink:** and **dataLinkManager:stopTrackingLink:** to follow the links in use.

Many applications do not need to track links individually, but there are several situations where it can be useful to know when a link is used. For example, the delegate may want to individually track links in order to continually update destination documents each time data for a link's source selection is modified; an individual link can be sent an **updateDestination** message whenever a modification is made that affects the destination.

Additionally, many links may be placed on the pasteboard when data is copied, but few of those links will actually ever get used. If the application must store selection-state information in the document, it should only do so for selections (and their associated links) that actually get used; this method is used to find out if the delegate wants to be informed when a link gets used.

importFile:at:

– **importFile:**(const char *)*filename* **at:**(NXSelection *)*selection*

If the application has added a link based on an entire file (that is, used **addLink:at:** to incorporate a link created by **initLinkedToFile:**), the delegate must implement this method to import the *filename* file at the destination described by *selection*. This method should return **self** upon success, or **nil** if the selection can't be resolved.

pasteFromPasteboard:at:

– **pasteFromPasteboard:**(Pasteboard *)*pasteboard* **at:**(NXSelection *)*selection*

If the application has added an ordinary destination link (that is, used **addLink:at:** to incorporate a link created by **initFromPasteboard:** or a related method), the delegate must implement this method to paste the updated data that has been made available on the pasteboard. The destination for the data is described by *selection*, which was supplied to the link manager as an argument to the **addLink:at:** method.

The data is read from the pasteboard just as it is for any ordinary paste; see the Pasteboard class specification for more information on reading data from a pasteboard. This method should return **self** upon success, or **nil** if the selection can't be resolved.

setSelection:

– **setSelection:**(NXSelection *)*selection*

Never invoked by the system.

showSelection:

– **showSelection:**(NXSelection *)*selection*

In an application that serves as a link source, the delegate should implement this method to show the source data for the specified selection *selection*. This method should scroll the document so the selected data is visible. It might additionally highlight the selected data using the function **NXFrameLinkRect()** with the argument *isDestination* set to NO. This method should return **self** upon success, or **nil** if the selection can't be resolved.

windowForSelection:

– **windowForSelection:**(NXSelection *)*selection*

In an application that serves as a link source, the delegate should implement this method to return the Window object for the given selection, or **nil** if the selection can't be resolved.

NXDataLinkPanel

Inherits From: Panel : Window : Responder : Object

Declared In: appkit/NXDataLinkPanel.h

Class Description

An NXDataLinkPanel is a Panel that allows the user to inspect data links. The NXDataLinkPanel functions primarily by sending messages to the current data link manager (representing the current document) and to the current link (representing the current selection if it's based on a data link). Thus, the panel should be informed, by a **setLink:andManager:isMultiple:** message, any time the selection changes or a document is created or activated. Since the selection may need to be tracked even before the panel is created, this message can be sent either to the NXDataLinkPanel class or the single instance.

The NXDataLinkPanel is generally displayed using Application's **orderFrontDataLinkPanel:** method. An application's sole instance of NXDataLinkPanel can be accessed with the **new** method.

Instance Variables

None declared in this class.

Method Types

Returning the panel	+ new + newContent:style:backing:buttonMask:defer:
Keeping the panel up to date	+ setLink:andManager:isMultiple: – setLink:andManager:isMultiple: + getLink:andManager:isMultiple: – getLink:andManager:isMultiple:
Customizing the panel	– setAccessoryView: – accessoryView
Responding to user input	– pickedBreakAllLinks: – pickedBreakLink: – pickedOpenSource: – pickedUpdateDestination: – pickedUpdateMode:

Class Methods

alloc

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

allocFromZone:

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

getLink:andManager:isMultiple:

+ **getLink:**(NXDataLink **)*link*
  andManager:(NXDataLinkManager **)*linkManager*
  isMultiple:(BOOL *)*flag*

Gets information about the NXDataLinkPanel's currently selected link. This method returns the link in *link*, the link manager in *linkManager*, and the multiple selection status in *flag*. Whenever a link is selected or deselected, this information must be set using **setLink:andManager:isMultiple:**. Returns **self**.

new

+ **new**

Returns the application's sole NXDataLinkPanel object, creating it if necessary.

newContent:style:backing:buttonMask:defer:

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

Initializes the NXDataLinkPanel object. You never invoke this method; use **new** instead.

setLink:andManager:isMultiple:

+ **setLink:**(NXDataLink *)*link*
  andManager:(NXDataLinkManager *)*linkManager*
  isMultiple:(BOOL)*flag*

Informs the NXDataLinkPanel of the current document and selection. This message must be sent any time data based on a data link is selected or deselected, or when a document (and therefore a new link manager) is activated. Since the state of the selection always needs to be tracked, this message can be sent to either the NXDataLinkPanel class or instance.

link is the currently selected link; it should be **nil** if no link is selected. *linkManager* is the current link manager. *flag* should be YES if the panel is to indicate that more than one link is selected. Returns **self**.

Instance Methods

accessoryView

– **accessoryView**

Returns the NXDataLinkPanel's custom accessory view, set by **setAccessoryView:**.

See also: – **setAccessoryView:**

getLink:andManager:isMultiple:

– **getLink:**(NXDataLink **)*link*
andManager:(NXDataLinkManager **)*linkManager*
isMultiple:(BOOL *)*flag*

Returns information about the NXDataLinkPanel's currently selected link. This method returns the link in *link*, the link manager in *linkManager*, and the multiple selection status in *flag*. This method functions identically to the class method of the same name. Whenever a link is selected or deselected, this information must be set using **setLink:andManager:isMultiple:**. Returns the NXDataLinkPanel class.

pickedBreakAllLinks:

– **pickedBreakAllLinks:***sender*

Invoked when the user clicks the Break All Links button, this method puts up an attention panel to confirm the user's action, and then sends a **breakAllLinks** message to the current link manager, as set by **setLink:andManager:isMultiple:**. Returns **self**.

See also: – **breakAllLinks** (NXDataLinkManager)

pickedBreakLink:

– **pickedBreakLink:***sender*

Invoked when the user clicks the Break Link button, this method puts up an attention panel to confirm the user's action, and then sends a **break** message to the current link, as set by **setLink:andManager:isMultiple:**. Returns **self**.

See also: – **break** (NXDataLink)

pickedOpenSource:

– **pickedOpenSource:***sender*

Invoked when the user clicks the Open Source button, this method sends a **openSource** message to the current link, as set by **setLink:andManager:isMultiple:**. Returns **self**.

See also: – **openSource** (NXDataLink)

pickedUpdateDestination:

– **pickedUpdateDestination:***sender*

Invoked when the user clicks the Update from Source button, this method sends a message to the current link to verify and update the data source and then update the destination data. Returns **self**.

See also: – **updateDestination** (NXDataLink), + **setLink:andManager:isMultiple:**

pickedUpdateMode:

– **pickedUpdateMode:***sender*

Invoked when the user selects the update mode, this method sends a **setUpdateMode:** message to the current link, as set by **setLink:andManager:isMultiple:**. Returns **self**.

See also: – **setUpdateMode:** (NXDataLink)

setAccessoryView:

– **setAccessoryView:***aView*

Adds *aView* to the NXDataLinkPanel's view hierarchy. Applications can invoke this method to add a View that contains their own controls. The 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 the panel's current accessory view, if any, is removed. Returns the old accessory view.

setLink:andManager:isMultiple:

– **setLink:**(NXDataLink *)*link*
andManager:(NXDataLinkManager *)*linkManager*
isMultiple:(BOOL)*flag*

Informs the NXDataLinkPanel of the current document and selection. This message must be sent any time data based on a data link is selected or deselected, or when a document (and therefore a new link manager) is activated. This method functions identically to the class method of the same name; since the state of the selection always needs to be tracked, this message can be sent to either the NXDataLinkPanel class or instance.

link is the currently selected link; it should be **nil** if no link is selected. *linkManager* is the current link manager. *flag* should be YES if the panel is to indicate that more than one link is selected. Returns the NXDataLinkPanel class.

NXEPSImageRep

Inherits From: NXImageRep : Object

Declared In: appkit/NXEPSImageRep.h

Class Description

An NXEPSImageRep is an object that can render an image from encapsulated PostScript code (EPS). The size of the object is set from the bounding box specified in the EPS header comments. Other information about the image should be supplied using inherited NXImageRep methods.

Like most other kinds of NXImageReps, an NXEPSImageRep is generally used indirectly, through an NXImage object.

Instance Variables

None declared in this class.

Method Types

Initializing a new NXEPSImageRep instance

- initFromSection:
- initFromFile:
- initFromStream:

Creating a List of NXEPSImageReps

+ newListFromSection:
+ newListFromSection:zone:
+ newListFromFile:
+ newListFromFile:zone:
+ newListFromStream:
+ newListFromStream:zone:

Copying and freeing an NXEPSImageRep	– copyFromZone: – free
Getting the rectangle that bounds the image	– getBoundingBox:
Getting image data	– getEPS:length:
Drawing the image	– prepareGState – drawIn: – draw
Archiving	– read: – write:

Instance Methods

newListFromFile:

+ (List *)**newListFromFile:**(const char *)*filename*

Creates one new NXEPSImageRep instance for each EPS image specified in the *filename* file, and returns a List object containing all the objects created. If no NXEPSImageReps can be created (for example, if *filename* doesn't exist or it doesn't contain EPS code or data that can be filtered to EPS), **nil** is returned. The List should be freed when it's no longer needed.

Each new NXEPSImageRep is initialized by the **initFromFile:** method, which reads a minimal amount of information about the image from the header comments in the file. The PostScript code will be read when it's needed to render the image.

The EPS format doesn't support more than one image per file. If *filename* contains EPS code, the List returned will be a list of one.

See also: + **newListFromFile:zone:**, – **initFromFile:**

newListFromFile:zone:

+ (List *)**newListFromFile:**(const char *)*filename* **zone:**(NXZone *)*aZone*

Returns a List of new NXEPSImageRep instances, just as **newListFromFile:** does, except that the NXEPSImageReps and the List object are allocated from memory located in *aZone*.

See also: + **newListFromFile:**, – **initFromFile:**

newListFromSection:

+ (List *)**newListFromSection:**(const char *)*name*

Creates one new NXEPSImageRep instance for each image specified in the *name* section of the __EPS segment in the executable file or in the *name* file in the application bundle, and returns a List object containing all the objects created. If not even one NXEPSImageRep can be created (for example, if the *name* section doesn't exist or it doesn't contain EPS code or data that can be filtered to EPS), **nil** is returned. The List should be freed when it's no longer needed.

Each new NXEPSImageRep is initialized by the **initFromSection:** method, which reads a minimal amount of information about the image from the EPS header comments. The PostScript code will be read only when it's needed to render the image.

The EPS format doesn't support more than one image per file. If the *name* section or file contains EPS code, the List returned will be a list of one.

See also: + **newListFromSection:zone:**, – **initFromSection:**

newListFromSection:zone:

+ (List *)**newListFromSection:**(const char *)*name* **zone:**(NXZone *)*aZone*

Returns a List of new NXEPSImageRep instances, just as **newListFromSection:** does, except that the List object and the NXEPSImageReps are allocated from memory located in *aZone*.

See also: + **newListFromSection:**, – **initFromSection:**

newListFromStream:

+ (List *)**newListFromStream:**(NXStream *)*stream*

Creates one new NXEPSImageRep instance for each image that can be read from *stream*, and returns a List object containing all the objects created. If not even one NXEPSImageRep can be created (for example, if the *stream* doesn't contain EPS code or data that can be converted to EPS), **nil** is returned. The List should be freed when it's no longer needed.

The data is read and each new object initialized by the **initFromStream:** method.

See also: + **newListFromStream:zone:**, – **initFromStream:**

newListFromStream:zone:

+ (List *)**newListFromStream:**(NXStream *)*stream* **zone:**(NXZone *)*aZone*

Returns a List of new NXEPSImageRep instances, just as **newListFromStream:** does, except that the List object and the NXEPSImageReps are allocated from memory located in *aZone*.

See also: + **newListFromStream:**, – **initFromStream:**

Instance Methods

copyFromZone:

– **copyFromZone:**(NXZone *)*zone*

Returns a new NXEPSImageRep instance that's an exact copy of the receiver. The new object will have its own copy of the image data. It doesn't need to be initialized. Both object and data are allocated from *zone*.

See also: – **copy** (Object)

draw

– (BOOL)**draw**

Draws the image at (0.0, 0.0) in the current coordinate system on the current device. This method returns YES if successful in rendering the image, and NO if not.

An NXEPSImageRep draws in a separate PostScript context and graphics state. Before the EPS code is interpreted, all graphics state parameters—with the exception of the CTM and device—are set to the Window Server's defaults and the defaults required by EPS conventions. If you want to change any of these defaults, you can do so by implementing a **prepareGState** method in an NXEPSImageRep subclass. The **draw** method invokes **prepareGState** just before sending the EPS code to the Window Server. For example, if you need to set a transfer function or halftone screen that's specific to the image, **prepareGState** is the place to do it.

See also: – **drawAt:** (NXImageRep), – **drawIn:**, – **prepareGState**

drawIn:

– (BOOL)**drawIn:**(const NXRect *)*rect*

Draws the image so that it fits inside the rectangle referred to by *rect*. The current coordinate system is translated and scaled so the image will appear at the right location and fit within the rectangle. The **draw** method is then invoked to produce the image. This method returns the value returned by the **draw** method, which indicates whether the image was successfully drawn.

The coordinate system is not restored after it has been altered.

See also: – **draw**, – **drawAt:** (NXImageRep)

free

– **free**

Deallocates the NXEPSImageRep.

getBoundingBox:

– **getBoundingBox:**(NXRect *)*theRect*

Provides the rectangle that bounds the image. The rectangle is copied from the "%%BoundingBox:" comment in the EPS header to structure referred to by *theRect*. Returns **self**.

getEPS:length:

– **getEPS:**(char **)*theEPS* **length:**(int *)*numBytes*

Sets the pointer referred to by *theEPS* so that it points to the EPS code. The length of the code in bytes is provided in the integer referred to by *numBytes*. Returns **self**.

init

Generates an error message. This method can't be used to initialize an NXEPSImageRep. Use one of the other **init**... methods instead.

See also: – **initFromSection:**, – **initFromFile:**, – **initFromStream:**

initFromFile:

– **initFromFile:**(const char *)*filename*

Initializes the receiver, a newly allocated NXEPSImageRep object, with the EPS image found in the *filename* file. Some information about the image is read from the EPS header comments, but the PostScript code won't be read until it's needed to render the image.

If the new object can't be initialized for any reason (for example, *filename* doesn't exist or doesn't contain EPS code), this method frees it and returns **nil**. Otherwise, it returns **self**.

This method is the designated initializer for NXEPSImageReps that read EPS code from a file.

See also: + **newListFromFile:**, – **initFromSection:**

initFromSection:

– **initFromSection:**(const char *)*name*

Initializes the receiver, a newly allocated NXEPSImageRep object, with the image found in the *name* section in the __EPS segment of the application executable or the *name* file in the application bundle. Some information about the image is read from the EPS header comments, but the PostScript code won't be read until it's needed to render the image.

If the new object can't be initialized for any reason (for example, the *name* section doesn't exist or doesn't contain EPS code), this method frees it and returns **nil**. Otherwise, it returns **self**.

This method is the designated initializer for NXEPSImageReps that read image data from the __EPS segment.

See also: + **newListFromSection:**, – **initFromFile:**

initFromStream:

– **initFromStream:**(NXStream *)*stream*

Initializes the receiver, a newly allocated NXEPSImageRep object, with the EPS image read from *stream*. If the new object can't be initialized for any reason (for example, *stream* doesn't contain EPS code), this method frees it and returns **nil**. Otherwise, it returns **self**.

This method is the designated initializer for NXEPSImageReps that read image data from a stream.

See also: + **newListFromStream:**

prepareGState

– **prepareGState**

Implemented by subclasses to initialize the graphics state before the image is drawn. The **draw** method sends a **prepareGState** message just before rendering the EPS code. This default implementation of the method does no initialization; it simply returns **self**.

See also: – **draw**

read:

– **read:**(NXTypedStream *)*stream*

Reads the NXEPSImageRep from the typed stream *stream*.

See also: – **write:**

write:

– **write:**(NXTypedStream *)*stream*

Writes the NXEPSImageRep to the typed stream *stream*.

See also: – **read:**

NXHelpPanel

Inherits From: Panel : Window : Responder : Object

Declared In: appkit/NXHelpPanel.h

Class Description

The NXHelpPanel class is the central component of the NeXTSTEP help system. It provides the Help panel that displays the text and illustrations that constitute your application's help information, and it stores associations of user-interface objects with specific passages of that text.

Users can display the Help panel by choosing the Help command from an application's Info menu. The panel employs the metaphor of a book: It displays a table of contents, body text, and an index. Users can browse through the text by clicking entries in the table of contents or index. The panel also supports hypertext-like help links, which appear as diamond-shaped images within the text and allow the user to easily follow cross references. By using the help cursor and clicking user-interface objects, the user can query the Help panel for information associated with those objects.

Adding the standard help system facilities to your application is easy. The Add Help Directory command in Project Builder supplies your application with a help directory and populates it with simple table-of-contents and index files. Interface Builder's Menu palette offers a Info submenu containing a preconfigured Help command. With this command and the two files, your application can display the generic help text that's available to all NeXTSTEP applications (for example, "Using the mouse" and "Sending a fax"). You can then add help information (as well as table-of-contents and index entries) that are specific to your application, and override any of the generic information your application inherits.

The Help Text

An NXHelpPanel object looks in a language-specific directory within the application's file package for the text that it will display. For example, if the user's language preference is English, the panel searches for a directory named **Help** within the **English.lproj** directory of the application's file package. It searches for two files: **TableOfContents.rtf** and **Index.rtfd**. There may also be one or more files containing the body text that the Help panel will display. The table-of-contents, index, and body files are interconnected by a system of *help links* and *help markers*.

A help marker is a named position holder in the stream of text—in most cases, it's invisible to users. A help link is a diamond-shaped button embedded in the text. Help links store a file name and, optionally, a help marker name. When a user clicks a help link, the Help panel displays the named file. If the help link also stores a marker name, the displayed file is scrolled to the position of the marker, and the text is selected from the marker's position to the end of the line.

The Text class provides the functionality for help links and markers, so this feature is available outside the Help panel. You use Edit to create and modify help documents. Edit's Help menu (accessible through the Format command) lets you insert links and markers and make the normally invisible help markers visible. Also in Edit, you can inspect and modify an existing help link or marker by holding down the Command key while clicking it.

Table-of-Contents and Index Files

The table-of-contents and index files are specially designed documents in Rich Text Format (RTF). An NXHelpPanel object identifies these files by name (**TableOfContents.rtf** and **Index.rtfd**) and processes them differently than it does other help files.

The table-of-contents file should contain one entry for each help text file in the help directory. Each entry begins with a help link that stores the name of the destination file for that entry. Following the link is the text of the entry, which may wrap and span several lines. Although the table of contents in the Help panel looks like it's displayed by a Matrix, it's actually displayed by a modified Text object. Thus, you can use the full generality of RTF to format your table of contents.

The index file is structured similarly although there is no enforced one-to-one mapping. Generally, the help link that begins an index entry stores both a file name and a marker name, since an index entry usually points to a specific word or phrase within a file.

Generic Help Files

The **Help** directory that Project Builder provides for a new application contains only table-of-contents and index files; no other help files are present. However, if you run the application, you'll find that its Help panel can display numerous help subjects, each of a general nature. This is because NeXTSTEP applications have access to the generic help information contained in **/usr/lib/NeXTSTEP/Resources/***language***.lproj/Help.store** (a compressed help directory).

When a help link is being resolved, the NXHelpPanel first looks for the specified file within the appropriate *language***.lproj/Help** directory of the application's file package. If the file isn't found, it then searches in **/usr/lib/NeXTSTEP/Resources/***language***.lproj/Help.store**. This search path is used for all links, whether they are in the table of contents, index, or body text. (Be forewarned: Edit doesn't apply this search path to help links, so if you open the table-of-contents or index files in Edit and click a help link, Edit will complain that the file can't be found, unless it exists within the application's file package.)

If one of these generic help files is inappropriate for your application, you have two remedies: You can remove the table-of-contents and index entries that refer to it, or you can override the file with one that's more appropriate. By placing a file of the same name and relative location within your application's **Help** directory, NXHelpPanel will display it rather than the generic file. See "Structure of the Generic Help Directory" below for the names and directory locations of the generic help files.

If you want to modify the generic file, use Interface Builder's Help Builder panel to display the file, and then select the entire text and copy and paste it into a new document. Save the document in your application's **Help** directory using the same name as shown in the Help Builder panel. Be sure to resize the window to the same width as the original so that the text will wrap to the same margins.

Associating Help Text with Objects

The NXHelpPanel class stores associations between user-interface objects and help text. When the user presses the Help modifier key (or, on older keyboards, simultaneously presses the Control and Alternate keys), a question mark cursor appears. If the user clicks an object using this cursor, the Help panel displays the associated help text.

The easiest way to create these associations is with Interface Builder's Help Builder panel. If your application has a **Help** directory containing the files **TableOfContents.rtf** and **Index.rtfd**, the Help Builder panel will let you use them to display the application's help files. By selecting an object in your application, displaying the appropriate help file in the Help Builder panel, and clicking the Attach File to Selection button, you establish the association.

You can also attach a help file to a user-interface object programmatically, by sending an **attachHelpFile:markerName:to:** message to the NXHelpPanel class object. This method takes a file name, a marker name, and an object **id** as its arguments. The **detachHelpFrom:** message removes such an association.

Just as with help links, an NXHelpPanel searches both the application's file package and the appropriate file in **/usr/lib/NeXTSTEP/Resources/***language***.lproj** in attempting to find the file associated with a particular user-interface object.

Hidden Files

Although in general there's a one-to-one relationship between table-of-contents entries and files in the **Help** directory, you can force a single table-of-contents entry to represent multiple "hidden" files. This can be useful in reducing the overall length of the table of contents. For example, Mail's Help panel contains a single entry, "Commands, buttons, and panels," that's highlighted no matter which user-interface object has been queried for its associated help information.

Hidden files can't be accessed from the table of contents; rather, the user must find them by Help-clicking an object in the application's user interface, by using the Help Panel's Find command, by using the Index, or by following a help link from some other file. However, when a hidden file is displayed, the Help panel must select some entry in the table of contents. In Mail, this entry is entitled "Commands, buttons, and panels."

Conversely, when the user selects such an table-of-contents entry, the Help panel must display one of the files in the directory of hidden files; by convention, this file must be named **Prolog.rtfd**. The prolog file for the "Commands, buttons, and panels" entry informs users that they can get help on any command, panel, or button by Help-clicking that object.

The table of contents supplied by Project Builder contains the entry "Commands" that corresponds to a directory of hidden files. Conceptually, these files exist in **/usr/lib/NeXTSTEP/Resources/***language***.lproj/Help/Objects** (in fact, they are contained in a compressed file derived from this original directory structure). In your application, you can add to (or override) these hidden files by creating an **Objects** subdirectory within your application's **Help** directory and placing the new files there.

The Help panel's Find button searches through all the files that are connected to table-of-contents entries, first looking in the application's **Help** directory and then in the generic help material. If your table of contents has a link to the **Objects** subdirectory (in other words, has the "Commands" entry, as provided by Project Builder), the hidden files will be searched too. If you don't want some hidden file in the generic help material to appear in your application's Help panel as the result of a Find operation, override the file with an empty file of the same name. Since the file is empty, no search string will ever be found in it, and it will effectively block the generic file of the same name from being searched.

Context-Sensitive Help

Your application can provide context-sensitive help; that is, the particular text displayed when the Help menu command is chosen or when the user Help-clicks an object can depend on the state of the application. Context-sensitive help requires the intervention of the Application object's delegate.

If the Application object's delegate responds to the **app:willShowHelpPanel:** message, it receives such a message just before the Help panel is displayed. Within the implementation of the **app:willShowHelpPanel:** method, the delegate can specify which file will appear in the panel:

```
- app:sender willShowHelpPanel:panel
{
    char path[MAXPATHLEN + 1];

    sprintf (path, "%s/%s", [panel helpDirectory],
      "Tasks/AddressingMail/CreatingAddressBook.rtfd");
    [panel showFile:path atMarker:NULL];
    return self;
}
```

The delegate must specify a fully qualified path since the NXHelpPanel object assumes that a partial path is relative to the currently displayed help file.

Indexing Help Text

The Help panel's Find button makes use of Indexing Kit facilities to quickly locate files containing the search string. (Note that the index discussed here is a binary file produced by the Indexing Kit, not the **Index.rtfd** file that contains the textual index).

To create an index for your help files, in a Terminal window switch to the **Help** directory you want indexed and then enter:

```
ixbuild -v
```

with no other arguments. The **ixbuild** program builds an index (named **.index.store**) for the **Help** directory and its subdirectories.

If your application contains a directory of "hidden" help files, you must first make an index of that directory before creating an index of the entire **Help** directory.

Help Supplements

Since in NeXTSTEP some applications have the ability to load executable modules dynamically (for example, a drawing program could allow the user to load a new drawing tool), an NXHelpPanel object provides the ability to load supplemental help information. When the application loads the module, it sends the NXHelpPanel object an **addSupplement:inPath:** message to inform the object of the location of the new help supplement. The NXHelpPanel object appends the contents of the supplement's **TableOfContents.rtf** to the existing table of contents, so the supplement should have a title that clearly sets it off from the main part of the table of contents, for example:

—PatternTool Supplement—

Pattern Options

Brick
Stucco
Wood
Tile
Custom

Resizing and Rotating

Blending Patterns

Index to Supplement

The supplement's index is only accessible from the table of contents; the Help panel's Index button displays the main index.

Structure of the Generic Help Directory

As mentioned earlier, the generic help text provided in NeXTSTEP is contained in the file **/usr/lib/NeXTSTEP/Resources/***language***.lproj/Help.store**. This compressed file was derived from a directory of help files. To override a generic help file, you'll need its name and location in the original directory structure. The following listing provides that information:

```
Help
    Index.rtfd
    Objects
        Menus
            Main
                ServicesMenu.rtfd
                WindowsMenu.rtfd
            Services
                OtherService.rtfd
            Windows
                ArrangeInFront.rtfd
                CloseWindow.rtfd
                MiniaturizeWindow.rtfd
                WindowName.rtfd
        Panels
            ColorsPanel.rtfd
            FaxPanel.rtfd
            FontPanel.rtfd
            LinkInspectorPanel.rtfd
            OpenPanel.rtfd
            PageLayoutPanel.rtfd
            PrintPanel.rtfd
            SavePanel.rtfd
            SpellingPanel.rtfd
            Prolog.rtfd
    TableOfContents.rtf
    Tasks
        GettingStarted
            AdjustBriteVolume.rtfd
            ButtonsSlidersFields.rtfd
            ChooseCommands.rtfd
            ClickingHelp.rtfd
            DetachSubmenu.rtfd
            FindingHelp.rtfd
            GettingHelpTopic.rtfd
            Scrolling.rtfd
            UsingMouse.rtfd
            WorkingWindows.rtfd
        Reference
            Cursor.rtfd
```

Instance Variables

None declared in this class.

Method Types

Initializing and freeing	+ new + newForDirectory: – addSupplement:inPath: – free
Attaching Help to objects	+ attachHelpFile:markerName:to: + detachHelpFrom:
Setting click-for-help	+ isClickForHelpEnabled + setClickForHelpEnabled:
Printing	– print: – printPanel:
Querying	– helpDirectory – helpFile
Showing help	– showFile:atMarker: – showHelpAttachedTo:

Class Methods

attachHelpFile:markerName:to:

+ **attachHelpFile:**(const char *)*filename*
  markerName:(const char *)*markerName*
  to:*anObject*

Associates *filename* and *markerName* with *anObject*. *filename* should be a path relative to the **Help** directory. *markerName* is the name of a marker within the file specified by *filename*. Returns **self**.

When *anObject* is Help-clicked, the Help panel displays the specified file, and the text is scrolled so that the point marked by *markerName* is visible. (If *markerName* is NULL, the file isn't scrolled.)

See also: – **detachHelpFrom:**

detachHelpFrom:

+ **detachHelpFrom:***anObject*

Removes any help information associated with *anObject*. Returns **self**.

See also: – **attachHelpFile:markerName:to:**

isClickForHelpEnabled

+ (BOOL)**isClickForHelpEnabled**

Returns whether Help-clicking is enabled.

See also: + **setClickForHelpEnabled:**

new

+ **new**

Creates, if necessary, and returns the NXHelpPanel object. This method invokes the **newForDirectory:** method, using "Help" as the single argument.

See also: + **newForDirectory:**

newForDirectory:

+ **newForDirectory:**(const char *)*helpDirectory*

Creates, if necessary, and returns the Help panel. If the panel is created, it loads the help directory specified by *helpDirectory*. The help directory must reside in the main bundle. If a Help panel already exists but has loaded a help directory other than *helpDirectory*, a second panel will be created.

See also: + **new**

setClickForHelpEnabled:

+ **setClickForHelpEnabled:**(BOOL)*enabled*

Sets whether Help-clicking is enabled. Normally most applications will leave this feature enabled. However on keyboards without a Help key, the user must hold the Control and Alternate modifiers while Help-clicking. Because some applications may depend upon the simultaneous use of these modifiers, they may need a way to disable the click-for-help feature. In this case, it is recommended that the application have a menu command to allow the user to toggle whether click-for-help is enabled or whether these modifiers are passed through for the application's use. The menu title should be "Disable Click for Help" and should toggle with "Enable Click for Help". Returns **self**.

See also: + **isClickForHelpEnabled**

Instance Methods

addSupplement:inPath:

– **addSupplement:**(const char *)*helpDirectory* **inPath:**(const char *)*supplementPath*

Adds supplemental help by appending the supplement's **TableOfContents.rtf** file to the existing table of contents. This method is designed to be used when an application dynamically loads a resource that has its own help information. Returns **self**.

free

– **free**

Frees the NXHelpPanel and its storage.

helpDirectory

– (NXAtom)**helpDirectory**

Returns the absolute path of the currently loaded help directory.

See also: – **helpFile**

helpFile

– (NXAtom)**helpFile**

Returns the path of the currently loaded help file relative to the current help directory.

See also: – helpDirectory

print:

– **print:***sender*

Prints the currently displayed help text and returns **self**.

printPanel:

– **printPanel:***sender*

This is the same as the **print:** method.

showFile:atMarker:

– **showFile:**(const char *)*filename* **atMarker:**(const char *)*markerName*

Causes the Help panel to display the help contained in *filename*. If *markerName* is non-NULL, then the marker is sought in the file. If found, it's scrolled into view and the text from the marker to the end of the line is highlighted. If the file is not a full path, then it's assumed to be relative to the currently displayed help file. Returns **self**.

showHelpAttachedTo:

– (BOOL)**showHelpAttachedTo:***anObject*

Causes the Help panel to display the help attached to *anObject*. Returns YES if the object has help attached to it, NO if not.

NXImage

Inherits From: Object

Declared In: appkit/NXImage.h

Class Description

An NXImage object contains an image that can be composited anywhere without first being drawn in any particular View. It manages the image by:

- Reading image data from the application bundle, from a Pasteboard, or from an NXStream.
- Choosing the representation that's appropriate for a particular data type.
- Keeping multiple representations of the same image.
- Choosing the representation that's appropriate for any given display device.
- Caching the representations it uses by rendering them in off-screen windows.
- Optionally retaining the data used to draw the representations, so that they can be reproduced when needed.
- Compositing the image from the off-screen cache to where it's needed on-screen.
- Reproducing the image for the printer so that it matches what's displayed on-screen, yet is the best representation possible for the printed page.
- Automatically using any filtering services installed by the user to convert image data from unsupported formats to supported formats.

Defining an Image

An image can be created from various types of data:

- Encapsulated PostScript code (EPS)
- Bitmap data in Tag Image File Format (TIFF)
- Untagged (raw) bitmap data
- RenderMan Interface Bytestream code (RIB)

- Other image data supported by an NXImageRep subclass registered with the NXImage class
- Data that can be filtered to a supported type by a user-installed filter service

If data is placed in a file (for example, in an application bundle), the NXImage object can access the data whenever it's needed to create the image. If data is read from a stream, the NXImage object may need to retain the data itself.

Images can also be defined by the program, in two ways:

- By drawing the image in an off-screen window maintained by the NXImage object. In this case, the NXImage maintains only the cached image.
- By defining a method that can be used to draw the image when needed. This allows the NXImage to delegate responsibility for producing the image to some other object.

Image Representations

An NXImage object can keep more than one representation of an image. Multiple representations permit the image to be customized for the display device. For example, different hand-tuned TIFF images can be provided for monochrome and color screens, and an EPS representation or a custom method might be used for printing. All representations are versions of the same image.

An NXImage returns a List of its representations in response to a **representationList** message. Each representation is a kind of NXImageRep object:

NXEPSImageRep	An image that can be recreated from EPS data that's either retained by the object or at a known location in the file system.
NXBitmapImageRep	An image that can be recreated from bitmap or TIFF data.
N3DRIBImageRep	An image that can be recreated from RIB data.
NXCustomImageRep	An image that can be redrawn by a method defined in the application.
NXCachedImageRep	An image that has been rendered in an off-screen cache from data or instructions that are no longer available. The image in the cache provides the only data from which the image can be reproduced.

You can define other NXImageRep subclasses for objects that render images from other types of source data. You make a subclass known to NXImage by invoking the **registerImageRep:** class method. NXImage determines the data types that each subclass can support by invoking its **imageUnfilteredFileTypes** and **imageUnfilteredPasteboardTypes** methods.

Choosing Representations

The NXImage object will choose the representation that best matches the rendering device. By default, the choice is made according to the following set of ordered rules. Each rule is applied in turn until the choice of representation is narrowed to one.

1. Choose a color representation for a color device, and a gray-scale representation for a monochrome device.

2. Choose a representation with a resolution that matches the resolution of the device, or if no representation matches, choose the one with the highest resolution.

   By default, any image representation with a resolution that's an integer multiple of the device resolution is considered to match. If more than one representation matches, the NXImage will choose the one that's closest to the device resolution. However, you can force resolution matches to be exact by passing NO to the **setMatchedOnMultipleResolution:** method.

   Rule 2 prefers TIFF and bitmap representations, which have a defined resolution, over EPS representations, which don't. However, you can use the **setEPSUsedOnResolutionMismatch:** method to have the NXImage choose an EPS representation in case a resolution match isn't possible.

3. If all else fails, choose the representation with a specified bits per sample that matches the depth of the device. If no representation matches, choose the one with the highest bits per sample.

By passing NO to the **setColorMatchPreferred:** method, you can have the NXImage try for a resolution match before a color match. This essentially inverts the first and second rules above.

If these rules fail to narrow the choice to a single representation—for example, if the NXImage has two color TIFF representations with the same resolution and depth—the one that will be chosen is system dependent.

Caching Representations

When first asked to composite the image, the NXImage object chooses the representation that's best for the destination display device, as outlined above. It renders the representation in an off-screen window on the same device, then composites it from this cache to the desired location. Subsequent requests to composite the image use the same cache. Representations aren't cached until they're needed for compositing.

When printing, the NXImage tries not to use the cached image. Instead, it attempts to render on the printer—using the appropriate image data, or a delegated method—the best version of the image that it can. Only as a last resort will it image the cached bitmap.

Image Size

Before an NXImage can be used, the size of the image must be set, in units of the base coordinate system. If a representation is smaller or larger than the specified size, it can be scaled to fit.

If the size of the image hasn't already been set when the NXImage is provided with a representation, the size will be set from the data. The bounding box is used to determine the size of an NXEPS ImageRep. The TIFF fields "ImageLength" and "ImageWidth" are used to determine the size of an NXBitmapImageRep. The **RiDisplay()** parameters are used to determine the size of an N3DRIBImageRep.

Coordinate Systems

Images have the horizontal and vertical orientation of the base coordinate system; they can't be rotated or flipped. When composited, an image maintains this orientation, no matter what coordinate system it's composited to. (The destination coordinate system is used only to determine the location of a composited image, not its size or orientation.)

It's possible to refer to portions of an image when compositing (or when defining subimages), by specifying a rectangle in the image's coordinate system, which is identical to the base coordinate system, except that the origin is at the lower left corner of the image.

Named Images

An NXImage object can be identified either by its **id** or by a name. Assigning an NXImage a name adds it to a database kept by the class object; each name in the database identifies one and only one instance of the class. When you ask for an NXImage object by name (with the **findImageNamed:** method), the class object returns the one from its database, which also includes all the system bitmaps provided by the Application Kit. If there's no object in the database for the specified name, the class object tries to create one by looking (for historical reasons) in the __ICON, __EPS, and __TIFF segments of the application's executable file, and then in the application bundle.

If a section or file matches the name, an NXImage is created from the data stored there. You can therefore create NXImage objects simply by including EPS or TIFF data for them within the executable file, or in files inside the application's file package.

The job of displaying an image within a View can be entrusted to a Cell object. A Cell identifies the image it's to display by the name of the NXImage object. The following code sets **myCell** to display one of the system bitmaps:

```
id myCell = [[Cell alloc] initIconCell:"NXswitch"];
```

Image Filtering Services

NXImage is designed to automatically take advantage of user-installed filter services for converting unsupported image file types to supported image file types. The class method **imageFileTypes** returns a list of all file types from which NXImage can create an instance of itself. This list includes all file types supported by registered subclasses of NXImageRep, and those types that can be converted to supported file types through a user-installed filter service.

Instance Variables

char ***name**;

name	The name assigned to the image.

Method Types

Initializing a new NXImage instance	– init – initSize: – initFromSection: – initFromFile: – initFromStream: – initFromPasteboard: – initFromImage:rect: – copyFromZone:
Freeing an NXImage object	– free
Setting the size of the image	– setSize: – getSize:
Referring to images by name	– setName: – name + findImageNamed:

Specifying the image
- useDrawMethod:inObject:
- useFromSection:
- useFromFile:
- loadFromFile:
- useRepresentation:
- useCacheWithDepth:
- loadFromStream:
- lockFocus
- lockFocusOn:
- unlockFocus

Using the image
- composite:toPoint:
- composite:fromRect:toPoint:
- dissolve:toPoint:
- dissolve:fromRect:toPoint:

Choosing which image representation to use
- setColorMatchPreferred:
- isColorMatchPreferred
- setEPSUsedOnResolutionMismatch:
- isEPSUsedOnResolutionMismatch
- setMatchedOnMultipleResolution:
- isMatchedOnMultipleResolution

Getting the representations
- lastRepresentation
- bestRepresentation
- representationList
- removeRepresentation:

Determining how the image is stored
- setUnique:
- isUnique
- setDataRetained:
- isDataRetained
- setCacheDepthBounded:
- isCacheDepthBounded
- getImage:rect:

Determining how the image is drawn
- setFlipped:
- isFlipped
- setScalable:
- isScalable
- setBackgroundColor:
- backgroundColor
- drawRepresentation:inRect:
- recache

Assigning a delegate
- setDelegate:
- delegate

Producing TIFF data for the image
- writeTIFF:
- writeTIFF:allRepresentations:
- writeTIFF:allRepresentations:usingCompression: andFactor:

Managing NXImageRep subclasses
+ registerImageRep:
+ unregisterImageRep:
+ imageRepForFileType:
+ imageRepForPasteboardType:
+ imageRepForStream:

Testing image data sources
+ canInitFromPasteboard:
+ imageFileTypes
+ imagePasteboardTypes

Archiving
- read:
- write:
- finishUnarchiving

Class Methods

canInitFromPasteboard:

+ (BOOL)**canInitFromPasteboard:**(Pasteboard *)*pasteboard*

Tests whether NXImage can create an instance of itself from the data represented by pasteboard. Returns YES if NXImage's list of registered NXImageReps includes a class that can handle the data represented by *pasteboard.*

By default, this method returns YES if *pasteboard*'s type is NXTIFFPboardType, NXPostScriptPboardType, or NXFilenamePboardType (for file names with extension

".tiff", ".tif", or ".eps"). Applications linked against libMedia_s.a also return YES if *pasteboard*'s type is N3DRIBPboardType or NXFilenamePboardType (for file names with extension ".rib").

NXImage uses the NXImageRep class method **imageUnfilteredPasteboardTypes** to find the class that can handle the data in *pasteboard*. When creating a subclass of NXImageRep that accepts image data from a non-default pasteboard type, override the **imageUnfilteredPasteboardTypes** method to notify NXImage of the pasteboard types your class supports.

See also: + imagePasteboardTypes, + imageRepForPasteboard:, + imageUnfilteredPasteboardTypes (NXImageRep), N3DRIBImageRep (3D Kit Classes)

findImageNamed:

+ findImageNamed:(const char *)*name*

Returns the NXImage instance associated with *name*. The returned object can be:

- One that's been assigned a name with the **setName:** method,
- One of the named system bitmaps provided by the Application Kit, or
- One that's been created and named by this method.

If there's no known NXImage with *name*, this method tries to create one by searching for image data in the application's executable file and in the application bundle:

1. For historical reasons, it looks in the application executable. It looks first in the __ICON segment for a *name* section containing either Encapsulated PostScript code (EPS) or Tag Image File Format (TIFF) data. It looks next for a section with TIFF data in the __TIFF segment if *name* includes a ".tiff" extension, or for a section containing EPS data in the __EPS segment if *name* includes a ".eps" extension. If *name* has neither extension, both segments are searched, first after adding the appropriate extension to *name*, then for *name* alone, without an extension. If it finds sections in both segments, it creates both EPS and TIFF representations of the image.

2. Next, it searches for *name* files in the **lproj** directories in the application's main bundle. It searches for all file types (extensions) handled by all registered NXImageReps; by default, the files searched for include those with the extension "tiff", "tif", and "eps". It searches the language directories that the user specified for this application, or (if none) those specified by the user's default language preferences (see Application's **systemLanguages** method).

3. Finally, if there's no file named *name* in the main bundle's relevant language directories, it looks for *name* files in the main bundle (but outside the **lproj** directories). Again, it searches for all file types handled by all registered NXImageReps.

If a section or file contains data for more than one image, a separate representation is created for each one. If an image representation can't be found for *name*, no object is created and **nil** is returned.

The preferred way to name an image is to ask for a *name* without the extension, but to include the extension on the section name or file name.

This method treats all images found in the __ICON segment as application or document icons, since the point of putting an image in that segment rather than in __TIFF or __EPS is to advertise it to the Workspace Manager. The Workspace Manager requires icons to be no more than 48 pixels wide by 48 pixels high. Therefore, an NXImage created from an __ICON section has its size set to 48.0 by 48.0 and is made scalable.

The image returned by this method should not be freed, unless it's certain that no other objects reference it.

See also: – **setName:**, – **name**, + **registerImageRep**, + **imageFileTypes**

imageFileTypes

+ (const char *const *)**imageFileTypes**

Returns a null-terminated array of strings representing file types for which a registered NXImageRep exists. This list includes all file types supported by registered subclasses of NXImageRep, and those types that can be converted to supported file types through a user-installed filter service. The array returned by this method may be passed directly to the OpenPanel's **runModalForTypes:** method. The returned array belongs to the system, and should not be freed by the application.

File types are identified by extension. By default, the list returned by this method contains "tiff", "tif", "eps", and, for applications that use the 3D Kit, "rib".

When creating a subclass of NXImageRep that accepts image data from non-default file types, override the **imageUnfilteredFileTypes** method to notify NXImage of the file types your class supports.

See also: + **imageRepForFiletype:**, + **imageUnfilteredFileTypes** (NXImageRep class)

imagePasteboardTypes

+ (const NXAtom *)**imagePasteboardTypes**

Returns a null-terminated list of pasteboard types for which a registered NXImageRep exists. This list includes all pasteboard types supported by registered subclasses of

NXImageRep, and those that can be converted to supported pasteboard types through a user-installed filter service. The returned list belongs to the system, and should not be freed by the application.

By default, the list returned by this method contains "NXPostScriptPboardType," "NXTIFFPboardType," and, for applications that use the 3D Kit, "N3DRIBPboardType."

When creating a subclass of NXImageRep that accepts image data from non-default pasteboard types, override the **imageUnfilteredPasteboardTypes** method to notify NXImage of the pasteboard types your class supports.

See also: + **imageRepForPasteboardtype:**,
+ **imageUnfilteredPasteboardTypes** (NXImageRep)

imageRepForFileType:

+ (Class)**imageRepForFileType:**(const char *)*type*

Returns the NXImageRep subclass that supports files of *type*. *type* represents the file extension, which represents the type of data in the file. By default, this method returns the NXBitmapImageRep class for a *type* of "tiff" or "tif" and the NXEPSImageRep class for a *type* of "eps". If you create a subclass of NXImageRep that supports other file types, you must override the NXImageRep **imageUnfilteredFileTypes** class method to return the extensions representing those file types, then register your subclass using NXImage's **registerImageRep:** class method.

See also: + **registerImageRep:**, – **imageFileTypes**,
– **imageUnfilteredFileTypes** (NXImageRep)

imageRepForPasteboardType:

+ (Class)**imageRepForPasteboardType:**(NXAtom)*type*

Returns the NXImageRep subclass that supports pasteboards of *type*. By default, this method returns NXBitmapImageRep for a *type* of NXTIFFPboardType and NXEPSImageRep for a *type* of NXPostscriptPboardType. If *type* is NXFilenamePboardType, this method returns the registered NXImageRep that supports files with the extension of the file name on the pasteboard, either directly or through a user-installed filter service. If no registered NXImageRep handles data of *type*, returns **nil**.

If you create a subclass of NXImageRep that supports other pasteboard types, you must override the NXImageRep class method **imageUnfilteredPasteboardTypes** to return the

extensions representing those pasteboard types, then register your subclass using NXImage's class method **registerImageRep:**.

See also: + **registerImageRep:**, – **imagePasteboardTypes**, – **imageUnfilteredPasteboardTypes** (NXImageRep class)

imageRepForStream:

+ (Class)**imageRepForStream:**(NXStream *)*stream*

Returns the NXImageRep subclass that can be instantiated from the data in *stream*. By default, this method returns NXBitmapImageRep for a *stream* containing TIFF data, NXEPSImageRep for a *stream* containing EPS data, and **nil** for a *stream* containing an unsupported data type. *stream* must be seekable.

If you create a subclass of NXImageRep that supports other data types, you must override the NXImageRep **canLoadFromStream:** class method to determine whether your subclass can be instantiated from data in *stream*. You must also register your subclass using NXImage's **registerImageRep:** class method.

See also: + **registerImageRep:**, – **canLoadFromStream:** (NXImageRep)

registerImageRep:

+ (void)**registerImageRep:***imageRepClass*

Informs NXImage of the existence of a subclass of NXImageRep.

This method adds the class to NXImage's list of image reps without verifying the behavior of *imageRepClass*. When the application calls NXImage class methods such as **imageFileTypes**, **imagePasteboardTypes**, and **imageRepForFileType**, the class checks this list to find the types of image data it can support. *imageRepClass* must implement the methods **imageUnfilteredFileTypes**, **imageUnfilteredPasteboardTypes**, and **canLoadFromStream:** to inform NXImage of the data types it supports.

This method may be invoked multiple times; *imageRepClass* is added to NXImage's list of image reps only the first time it is invoked.

See also: + **imageFileTypes**, + **imagePasteboardTypes**, + **imageRepForFileType**, NXImageRep class description

unregisterImageRep:

+ (void)**unregisterImageRep:***imageRepClass*

Informs NXImage of the resignation of a subclass of NXImageRep.

See also: + registerImageRep:

Instance Methods

backgroundColor

– (NXColor)**backgroundColor**

Returns the background color of the rectangle where the image is cached. If no background color has been specified, NX_COLORCLEAR is returned, indicating a totally transparent background.

The background color will be visible when the image is composited only if the image doesn't completely cover all the pixels within the area specified for its size.

See also: – **setBackgroundColor:**

bestRepresentation

– (NXImageRep *)**bestRepresentation**

Returns the image representation that best matches the display device with the deepest frame buffer currently available to the Window Server.

See also: – **representationList**

composite:fromRect:toPoint:

– **composite:**(int)*op*
fromRect:(const NXRect *)*aRect*
toPoint:(const NXPoint *)*aPoint*

Composites the area enclosed by the *aRect* rectangle to the location specified by *aPoint* in the current coordinate system. The *op* and *aPoint* arguments are the same as for **composite:toPoint:**. The source rectangle is specified relative to a coordinate system that has its origin at the lower left corner of the image, but is otherwise the same as the base coordinate system.

This method doesn't check to be sure that the rectangle encloses only portions of the image. Therefore, it can conceivably composite areas that don't properly belong to the image, if the *aRect* rectangle happens to include them. If this turns out to be a problem, you can prevent it from happening by having the NXImage cache its representations in their own individual windows (with the **setUnique:** method). In this case, the window's clipping path will prevent anything but the image from being composited.

Compositing part of an image is as efficient as compositing the whole image, but printing just part of an image is not. When printing, it's necessary to draw the whole image and rely on a clipping path to be sure that only the desired portion appears.

If successful in compositing (or printing) the image, this method returns **self**. If not, it returns **nil**.

See also: – **composite:toPoint:**, – **setUnique:**

composite:toPoint:

– **composite:**(int)*op* **toPoint:**(const NXPoint *)*aPoint*

Composites the image to the location specified by *aPoint*. The first argument, *op*, names the type of compositing operation requested. It should be one of the following constants:

NX_CLEAR	NX_SOVER	NX_DOVERNX_XOR
NX_COPY	NX_SIN	NX_DIN
NX_PLUSD	NX_SOUT	NX_DOUT
NX_PLUSL	NX_SATOP	NX_DATOP

aPoint is specified in the current coordinate system—the coordinate system of the currently focused View—and designates where the lower left corner of the image will appear. The image will have the orientation of the base coordinate system, regardless of the destination coordinates.

The image is composited from its off-screen window cache. Since the cache isn't created until the image representation is first used, this method may need to render the image before compositing.

When printing, the compositing methods do not composite, but attempt to render the same image on the page that compositing would render on the screen, choosing the best available representation for the printer. The *op* argument is ignored.

If successful in compositing (or printing) the image, this method returns **self**. If not, it returns **nil**.

See also: – **composite:fromRect:toPoint:**, – **dissolve:toPoint:**

copyFromZone:

– **copyFromZone:**(NXZone *)*zone*

Returns a new instance of NXImage that's an exact copy of the receiver. Memory for the new instance is allocated from *zone*. Cached image reps are copied fully, the new NXImage has its own copy of the image data. Lazily allocated image reps are copied with only source information (for example, file names); the image can be recreated from this source *when the NXImage is asked to composite itself.*

See also: – **copyFromZone:** (NXCachedImageRep)

delegate

– **delegate**

Returns the delegate of the NXImage object, or **nil** if no delegate has been set.

See also: – **setDelegate:**

dissolve:fromRect:toPoint:

– **dissolve:**(float)*delta*
fromRect:(const NXRect *)*aRect*
toPoint:(const NXPoint *)*aPoint*

Composites the *aRect* portion of the image to the location specified by *aPoint*, just as **composite:fromRect:toPoint:** does, but uses the **dissolve** operator rather than **composite**. *delta* is a fraction between 0.0 and 1.0 that specifies how much of the resulting composite will come from the NXImage. If the source image contains alpha, this operation may promote the destination Window.

When printing, this method is identical to **composite:fromRect:toPoint:**. The *delta* argument is ignored.

If successful in compositing (or printing) the image, this method returns **self**. If not, it returns **nil**.

See also: – **dissolve:toPoint:**, – **composite:fromRect:toPoint:**

dissolve:toPoint:

– **dissolve:**(float)*delta* **toPoint:**(const NXPoint *)*aPoint*

Composites the image to the location specified by *aPoint*, just as **composite:toPoint:** does, but uses the **dissolve** operator rather than **composite**. *delta* is a fraction between 0.0 and 1.0 that specifies how much of the resulting composite will come from the NXImage. If the source image contains alpha, this operation may promote the destination Window.

To slowly dissolve one image into another, this method (or **dissolve:fromRect:toPoint:**) needs to be invoked repeatedly with an ever-increasing *delta*. Since *delta* refers to the fraction of the source image that's combined with the original destination (not the destination image after some of the source has been dissolved into it), the destination image should be replaced with the original destination before each invocation. This is best done in a buffered window before the results of the composite are flushed to the screen.

When printing, this method is identical to **composite:toPoint:**. The *delta* argument is ignored.

If successful in compositing (or printing) the image, this method returns **self**. If not, it returns **nil**.

See also: – dissolve:fromRect:toPoint:, – composite:toPoint:

drawRepresentation:inRect:

– (BOOL)**drawRepresentation:**(NXImageRep *)*imageRep*
inRect:(const NXRect *)*rect*

Fills the specified rectangle with the background color, then sends the *imageRep* a **drawIn:** message to draw itself inside the rectangle (if the NXImage is scalable), or a **drawAt:** message to draw itself at the location of the rectangle (if the NXImage is not scalable). The rectangle is located in the current window and is specified in the current coordinate system.

This method shouldn't be called directly; the NXImage uses it to cache and print its representations. By overriding it in a subclass, you can change how representations appear in the cache, and thus how they'll appear when composited. For example, your version of the method could scale or rotate the coordinate system, then send a message to **super** to perform this version.

This method returns the value returned by the **drawIn:** or **drawAt:** method, which indicates whether or not the representation was successfully drawn. When NO is returned, the NXImage will ask another representation, if there is one, to draw the image.

If the background color is fully transparent and the image is not being cached by the NXImage, the rectangle won't be filled before the representation draws.

See also: – **drawIn** (NXImageRep), – **drawAt:** (NXImageRep)

finishUnarchiving

– **finishUnarchiving**

Registers the name of the newly unarchived receiver, if it has a name, and returns **nil**. It also returns **nil** if the receiving NXImage doesn't have a name. However, if the receiver has a name that can't be registered because it's already in use, this method frees the receiver and returns the existing NXImage with that name, thus replacing the unarchived object with one that's already in use.

finishUnarchiving messages are generated automatically (by **NXReadObject**()) after the object has be unarchived (by **read:**) and initialized (by **awake**).

free

– **free**

Deallocates the NXImage and all its representations. If the object had been assigned a name, the name is removed from the class database.

Images that are obtained through **findImageNamed:** should not be freed unless it's certain that no other part of the program has similarly obtained a reference to the same object.

See also: + **findImageNamed:**

getImage:rect:

– **getImage:**(NXImage **)*theImage* **rect:**(NXRect *)*theRect*

Provides information about the receiving NXImage object, if it's a subimage of another NXImage. The parent NXImage is assigned to the variable referred to by *theImage*, and the rectangle where the receiver is located in that NXImage is copied into the structure referred to by *theRect*.

If the receiver is not a subimage of another NXImage object (if it wasn't initialized by **initFromImage:rect:**), the variable referred to by *theImage* is set to **nil** and the rectangle is not modified.

Returns **self**.

See also: – **initFromImage:rect:**

getSize:

– **getSize:**(NXSize *)*theSize*

Copies the size of the image into the structure specified by *theSize*. If no size has been set, all values in the structure will be set to 0.0. Returns **self**.

See also: – **setSize:**

init

– **init**

Initializes the receiver, a newly allocated NXImage instance, but does not set the size of the image. The size must be set, and at least one image representation provided, before the NXImage object can be used. The size can be set either through a **setSize:** message or by providing a representation that specifies a size.

See also: – **initSize:**, – **setSize:**

initFromFile:

– **initFromFile:**(const char *)*filename*

Initializes the receiver, a newly allocated NXImage instance, for the file *filename*. This method initializes lazily: the NXImage doesn't actually open *filename* or create an image representation from its data until an application attempts to composite or requests information about the NXImage. (Use the method **loadFromFile:** to immediately create an image representation for the data in a file.)

filename may be a full or relative pathname, and should include an extension that identifies the data type in the file. The mechanism that actually creates the image representation for *filename* will look for an NXImageRep subclass that handles that data type from among those registered with NXImage. By default, the files handled are those with the extensions "tiff", "tif", and "eps".

After finishing the initialization, this method returns **self**. However, if the new instance can't be initialized, it's freed and **nil** is returned. Since this method doesn't actually create an image representation for the data, your application should do error checking before attempting to use the image; one way to do so is by invoking the **lockFocus** method to check whether the image can be drawn.

This method uses the **useFromFile:** method to register *filename*. It's equivalent to a combination of **init** and **useFromFile:**.

See also: **– useFromFile:**, **– initSize:**, **– loadFromFile:**, **– loadFromFile:**, **+ imageRepForFileType:**, **+ registerImageRep**

initFromImage:rect:

– initFromImage:(NXImage *)*image* **rect:**(const NXRect *)*rect*

Initializes the receiver, a newly allocated NXImage instance, so that it's a subimage for the *rect* portion of another NXImage object, *image*. The size of the new object is set from the size of the *rect* rectangle. Returns **self**.

Once initialized, the new instance can't be altered and will remain dependent on the original image. Changes made to the original will also change the subimage.

Subimages should be used only as a way of avoiding **composite:fromRect:toPoint:** and **dissolve:fromRect:toPoint:** messages. They permit you to divide a large image into sections and assign each section a name. The name can then be passed to those Button and Cell methods that identify images by name rather than **id**.

See also: **– getImage:rect:**, **– initSize:**

initFromPasteboard:

– initFromPasteboard:(Pasteboard *)*pasteboard*

Initializes and returns the receiver, a newly allocated NXImage instance, from *pasteboard*. *pasteboard* should be of a type returned by one of the registered NXImageRep's **imageUnfilteredPasteboardTypes** methods; the default types supported are NXPostscriptPboardType (NXEPSImageRep) and NXTIFFPboardType (NXBitmapImageRep). If *pasteboard* is an NXFilenamePboardType, the file name should have an extension returned by one of the registered NXImageRep's **imageUnfilteredFileTypes** methods; the default types supported are "tiff", "tif", (NXBitmapImageRep) and "eps" (NXEPSImageRep).

If the data type on the pasteboard isn't supported by a registered NXImageRep, this method frees the receiver and returns **nil**. Otherwise this method invokes **initFromStream** to initialize the image.

See also: **+ registerImageRep:**

initFromSection:

– **initFromSection:**(const char *)*name*

Initializes the receiver, a newly allocated NXImage instance, on a resource *name* in the application directory. This method initializes lazily: the NXImage doesn't actually create a representation for the data in *name* until an application attempts to composite or requests information about the NXImage. When it does create a representation for *name*, the NXImage will look for data in the application's bundle.

For historical reasons, the mechanism that actually creates the image representation first looks for *name* in the application's executable file. It looks for a section with TIFF data in the __TIFF segment if *name* includes a ".tiff" extension, or for a section containing EPS data in the __EPS segment if *name* includes a ".eps" extension. If *name* has neither extension, both segments are searched, first after adding the appropriate extension to *name*, then for *name* alone, without an extension. If it finds sections in both segments, it creates both EPS and TIFF representations of the image.

Next, the mechanism searches for *name* files in the **lproj** directories in the application's main bundle. It searches for all file types (extensions) handled by all registered NXImageReps; by default, the files searched for include those with the extension "tiff", "tif", and "eps". It searches the language directories that the user specified for this application, or (if none) those specified by the user's default language preferences (see the Application class **systemLanguages** method). If a file *name* is found with no extension, the mechanism opens the file and looks for a registered subclass of NXImageRep that can handle the data (using the NXImage **imageRepForStream:** class method).

If a section contains EPS or TIFF data for more than one version of the image, a representation will be created and added to the NXImage for each image specified. If an application bundle contains more than one file named *name* (each with a different extension), a representation will be created and added to the NXImage for each file whose data type is supported by a registered subclass of NXImageRep. The size of the NXImage is set from image representation data.

After finishing the initialization, this method returns **self**. However, if the new instance can't be initialized, it's freed and **nil** is returned.

This method uses the **useFromSection:** method to register *name*. It's equivalent to a combination of **init** and **useFromSection:**.

See also: – **useFromSection:**, – **initSize:**

initFromStream:

– **initFromStream:**(NXStream *)*stream*

Initializes the receiver, a newly allocated NXImage instance, with the image or images specified in the data read from *stream*, and returns **self**. If the receiver can't be initialized for any reason, it's freed and **nil** is returned.

Since this method must store the data read from the stream or render the specified image immediately, it's less preferred than **initFromSection:** or **initFromFile:**, which can wait until the image is needed.

The stream must be seekable and should contain image data recognizable to a registered NXImageRep subclass. It's read using the **loadFromStream:** method, which will set the size of the NXImage from information found in the representation data. This method is equivalent to a combination of **init** and **loadFromStream:**.

See also: – **loadFromStream:**, – **initSize:**, + **registerImageRep:**, + **imageRepForStream:**, + **canLoadFromStream:** (NXImageRep)

initSize:

– **initSize:**(const NXSize *)*aSize*

Initializes the receiver, a newly allocated NXImage instance, to the size specified and returns **self**. The size should be specified in units of the base coordinate system. It must be set before the NXImage can be used.

This method is the designated initializer for the class (the method that incorporates the initialization of classes higher in the hierarchy through a message to **super**). All other **init...** methods defined in this class work through this method.

See also: – **setSize:**

isCacheDepthBounded

– (BOOL)**isCacheDepthBounded**

Returns YES if the depth of off-screen windows where the NXImage's representations are cached is bounded by the application's default depth limit, and NO if the depth of the caches can exceed that limit. The default is YES.

See also: – **setCacheDepthBounded:**, + **defaultDepthLimit** (Window)

isColorMatchPreferred

– (BOOL)**isColorMatchPreferred**

Returns YES if, when selecting the representation it will use, the NXImage first looks for one that matches the color capability of the rendering device (choosing a gray-scale representation for a monochrome device and a color representation for a color device), then if necessary narrows the selection by looking for one that matches the resolution of the device. If the return is NO, the NXImage first looks for a representation that matches the resolution of the device, then tries to match the representation to the color capability of the device. The default is YES.

See also: – **setColorMatchPreferred:**

isDataRetained

– (BOOL)**isDataRetained**

Returns YES if the NXImage retains the data needed to render the image, and NO if it doesn't. The default is NO. If the data is available in a section of the application executable or in a file that won't be moved or deleted, or if responsibility for drawing the image is delegated to another object with a custom method, there's no reason for the NXImage to retain the data. However, if the NXImage reads image data from a stream, you may want to have it keep the data itself; for example, to render the same image on another device at a different resolution.

See also: – **setDataRetained:**, – **loadFromStream:**

isEPSUsedOnResolutionMismatch

– (BOOL)**isEPSUsedOnResolutionMismatch**

Returns YES if an EPS representation of the image should be used whenever it's impossible to match the resolution of the device to the resolution of another representation of the image (a TIFF representation, for example). By default, this method returns NO to indicate that EPS representations are not necessarily preferred.

See also: – **setEPSUsedOnResolutionMismatch:**

isFlipped

– (BOOL)**isFlipped**

Returns YES if a flipped coordinate system is used when locating the image, and NO if it isn't. The default is NO.

See also: – setFlipped:

isMatchedOnMultipleResolution

– (BOOL)**isMatchedOnMultipleResolution**

Returns YES if the resolution of the device and the resolution specified for the image are considered to match if one is a multiple of the other, and NO if device and image resolutions are considered to match only if they are exactly the same. The default is YES.

See also: – setMatchedOnMultipleResolution:

isScalable

– (BOOL)**isScalable**

Returns YES if image representations are scaled to fit the size specified for the NXImage. If representations are not scalable, this method returns NO. The default is NO.

Representations created from data that specifies a size (for example, the "ImageLength" and "ImageWidth" fields of a TIFF representation or the bounding box of an EPS representation) will have the size the data specifies, which may differ from the size of the NXImage.

See also: – setScalable:

isUnique

– (BOOL)**isUnique**

Returns YES if each representation of the image is cached alone in an off-screen window of its own, and NO if they can be cached in off-screen windows together with other images. A return of NO doesn't mean that the windows are, in fact, shared, just that they can be. The default is NO.

See also: – setUnique:

lastRepresentation

– (NXImageRep *)**lastRepresentation**

Returns the last representation that was specified for the image (the last one added with methods like **useCacheWithDepth:**, **loadFromStream:**, and **initFromStream:**). If the NXImage has no representations, this method returns **nil**.

See also: – representationList, – bestRepresentation

loadFromFile:

– (BOOL)**loadFromFile:**(const char *)*filename*

Creates an image representation from the data read from *filename* and adds it to the receiving NXImage's list of representations. This method is equivalent to mapping the file to memory, then invoking **loadFromStream:**.

filename may be a full or relative pathname, and should include an extension that identifies the data type in the file. This method looks for an NXImageRep subclass that handles that data type from among those registered with NXImage. By default, the files handled are those with the extensions "tiff", "tif", and "eps". If the file name has no extension, or the extension is one not recognized by one of the registered subclasses of NXImageRep, this method opens a stream on the file and attempts to find a registered subclass of NXImageRep that can handle the data in the stream. If a file containing TIFF or EPS data includes more than one image, a separate representation is created for each one.

If the NXImage object doesn't retain image data (**isDataRetained** returns NO), the image will be rendered in an off-screen window and the representations will be of type NXCachedImageRep. If the data is retained, the representations will be of type NXBitmapImageRep or NXEPSImageRep, depending on the data.

If successful in creating at least one representation, this method returns YES. If not, it returns NO.

See also: – initFromStream:, + registerImageRep:, + imageRepForFileType:

loadFromStream:

– (BOOL)**loadFromStream:**(NXStream *)*stream*

Creates an image representation from the data read from *stream* and adds it to the receiving NXImage's list of representations. The stream must be seekable, and the data must be of a

type recognized by a registered NXImageRep. If the size of the NXImage hasn't yet been set, it will be set from information found in the TIFF fields or from the EPS bounding box comment. If the stream contains data specifying more than one image, a separate representation is created for each one.

If the NXImage object doesn't retain image data (**isDataRetained** returns NO), the image will be rendered in an off-screen window and the representations will be of type NXCachedImageRep. If the data is retained, the representations will be of type NXBitmapImageRep or NXEPSImageRep, depending on the data.

If successful in creating at least one representation, this method returns YES. If not, it returns NO.

See also: – **initFromStream:**, + **imageRepForStream:**

lockFocus

– (BOOL)**lockFocus**

Focuses on the best representation for the NXImage, by making the off-screen window where the representation will be cached the current window and a coordinate system specific to the area where the image will be drawn the current coordinate system. The best representation is the one that best matches the deepest available frame buffer; it's the same object returned by the **bestRepresentation** method.

If the NXImage has no representations, **lockFocus** creates one with the **useCacheWithDepth:** method, specifying the best depth for the deepest frame buffer currently in use. To add additional representations, **useCacheWithDepth:** messages must be sent explicitly.

This method returns YES if it's successful in focusing on the representation, and NO if not. A successful **lockFocus** message must be balanced by a subsequent **unlockFocus** message to the same NXImage. These messages bracket the code that draws the image.

lockFocus returns NO when the image can't be drawn; for example, because the file from which it was initialized is non-existent, or the data in that file is invalid. In this case, **lockFocus** will not have altered the current graphics state and should not be balanced by an **unlockFocus** message (in this regard, the NXImage **lockFocus** method differs from that of View).

See also: – **lockFocusOn:**, – **lockFocus** (View), – **unlockFocus**, – **useCacheWithDepth:**, – **bestRepresentation**

lockFocusOn:

– (BOOL)**lockFocusOn:**(NXImageRep *)*imageRep*

Focuses on the *imageRep* representation, by making the off-screen window where it will be cached the current window and a coordinate system specific to the area where the image will be drawn the current coordinate system.

This method returns YES if it's successful in focusing on the representation, and NO if it's not. A successful **lockFocusOn:** message must be balanced by a subsequent **unlockFocus** message to the same receiver. These messages bracket the code that draws the image. The **useCacheWithDepth:** method will add a representation specifically for this purpose. For example:

```
[myNXImage useCacheWithDepth:NX_TwoBitGrayDepth];
if ( [myNXImage lockFocusOn:[myImage lastRepresentation]] ) {
    /* drawing code goes here */
    [myNXImage unlockFocus];
}
```

If **lockFocusOn:** returns NO, it will not have altered the current graphics state and should not be balanced by an **unlockFocus** message.

See also: – **lockFocus**, – **lockFocus** (View), – **unlockFocus**, – **lastRepresentation**

name

– (const char *)**name**

Returns the name assigned to the NXImage, or NULL if no name has been assigned.

See also: – **setName:**, + **findImageNamed:**

read:

– **read:**(NXTypedStream *)*stream*

Reads the NXImage and all its representations from the typed stream *stream*.

See also: – **write:**

recache

– **recache**

Invalidates the off-screen caches of all representations and frees them. The next time any representation is composited, it will first be asked to redraw itself in the cache. NXCachedImageReps aren't destroyed by this method.

If an image is likely not to be used again, it's a good idea to free its caches, since that will reduce that amount of memory consumed by your program and therefore improve performance.

Returns **self**.

removeRepresentation:

– **removeRepresentation:**(NXImageRep *)*imageRep*

Frees the *imageRep* representation after removing it from the NXImage's list of representations. Returns **self**.

See also: – **representationList**

representationList

– (List *)**representationList**

Returns the List object containing all the representations of the image. The List belongs to the NXImage object, and there's no guarantee that the same List object will be returned each time. Therefore, rather than saving the object that's returned, you should ask for it each time you need it.

See also: – **bestRepresentation**, – **lastRepresentation**

setBackgroundColor:

– **setBackgroundColor:**(NXColor)*aColor*

Sets the background color of the image. The default is NX_COLORCLEAR, indicating a totally transparent background. The background color will be visible only for representations that don't completely cover all the pixels within the image when drawing. The background color is ignored for cached image representations (such as those created with **useCacheWithDepth:**); such caches are always created with a white background. This method doesn't cause the receiving NXImage to recache itself. Returns **self**.

See also: – **backgroundColor**

setCacheDepthBounded:

– **setCacheDepthBounded:**(BOOL)*flag*

Determines whether the depth of the off-screen windows where the NXImage's representations are cached should be limited by the application's default depth limit. If *flag* is NO, window depths will be determined by the specifications of the representations, rather than by the current display devices. The default is YES. This method doesn't cause the receiving NXImage to recache itself. Returns **self**.

See also: – **isCacheDepthBounded**, + **defaultDepthLimit** (Window)

setColorMatchPreferred:

– **setColorMatchPreferred:**(BOOL)*flag*

Determines how the NXImage will select which representation to use. If *flag* is YES, it first tries to match the representation to the color capability of the rendering device (choosing a color representation for a color device and a gray-scale representation for a monochrome device), and then if necessary narrows the selection by trying to match the resolution of the representation to the resolution of the device. If *flag* is NO, the NXImage first tries to match the representation to the resolution of the device, and then tries to match it to the color capability of the device. The default is YES. Returns **self**.

See also: – **isColorMatchPreferred**

setDataRetained:

– **setDataRetained:**(BOOL)*flag*

Determines whether the NXImage retains the data needed to render the image. The default is NO. If the data is available in a section of the application executable or in a file that won't be moved or deleted, or if responsibility for drawing the image is delegated to another object with a custom method, there's no reason for the NXImage to retain the data. However, if the NXImage reads image data from a stream, you may want to have it keep the data itself. Generally, this is useful to redraw the image to a device of different resolution.

If an image representation is created lazily (through the **useFromFile:** or **useFromSection:** methods), the only data retained is the source name.

See also: – **isDataRetained**

setDelegate:

– **setDelegate:***anObject*

Makes *anObject* the delegate of the NXImage. Returns **self**.

See also: – **delegate**

setEPSUsedOnResolutionMismatch:

– **setEPSUsedOnResolutionMismatch:**(BOOL)*flag*

Determines whether EPS representations will be preferred when there are no representations that match the resolution of the device. The default is NO. Returns **self**.

See also: – **isEPSUsedOnResolutionMismatch**

setFlipped:

– **setFlipped:**(BOOL)*flag*

Determines whether the polarity of the y-axis is inverted when drawing an image. If flag is YES, the image will have its coordinate origin in the upper left corner and the positive y-axis will extend downward. This method affects only the coordinate system used to draw the image, whether through a method assigned with the **useDrawMethod:object:** method or directly by focusing on a representation. It doesn't affect the coordinate system for specifying portions of the image for methods like **composite:fromRect:toPoint:** or **initFromImage:rect:**. This method doesn't cause the receiving NXImage to recache itself. Returns **self**.

See also: – **isFlipped**

setMatchedOnMultipleResolution:

– **setMatchedOnMultipleResolution:**(BOOL)*flag*

Determines whether image representations with resolutions that are exact multiples of the resolution of the device are considered to match the device. The default is YES. Returns **self**.

See also: – **isMatchedOnMultipleResolution**

setName:

– (BOOL)**setName:**(const char *)*string*

Sets *string* to be the name of the NXImage object and registers it under that name. If the object already has a name, that name is discarded. If *string* is already the name of another object or if the receiving NXImage is one of the system bitmaps provided by the Application Kit, the assignment fails.

If successful in naming or renaming the receiver, this method returns YES. Otherwise it returns NO.

See also: + **findImageNamed:**, – **name**

setScalable:

– **setScalable:**(BOOL)*flag*

Determines whether representations with sizes that differ from the size of the NXImage will be scaled to fit. The default is NO.

Generally, representations that are created through NXImage methods (such as **useCacheWithDepth:** or **initFromSection:**) have the same size as the NXImage. However, a representation that's added with the **useRepresentation:** method may have a different size, and representations created from data that specifies a size (for example, the "ImageLength" and "ImageWidth" fields of a TIFF representation or the bounding box of an EPS representation) will have the size specified.

This method doesn't cause the receiving NXImage to recache itself when it is next composited. Returns **self**.

See also: – **isScalable**

setSize:

– **setSize:**(const NXSize *)*aSize*

Sets the width and height of the image. The size referred to by *aSize* should be in units of the base coordinate system. The size of an NXImage must be set before it can be used. Returns **self**.

The size of an NXImage can be changed after it has been used, but changing it invalidates all its caches and frees them. When the image is next composited, the selected representation will draw itself in an off-screen window to recreate the cache.

See also: – **getSize:**, – **initSize:**

setUnique:

– **setUnique:**(BOOL)*flag*

Determines whether each image representation will be cached in its own off-screen window or in a window shared with other images. If *flag* is YES, each representation is guaranteed to be in a separate window. If *flag* is NO, a representation can be cached together with other images, though in practice it might not be. The default is NO.

If an NXImage is to be resized frequently, it's more efficient to cache its representations in unique windows.

This method does not invalidate any existing caches. Returns **self**.

See also: – **isUnique**

unlockFocus

– **unlockFocus**

Balances a previous **lockFocus** or **lockFocusOn:** message. All successful **lockFocus** and **lockFocusOn:** messages (those that return YES) must be followed by a subsequent **unlockFocus** message. Those that return NO should never be followed by **unlockFocus**.

Returns **self**.

See also: – **lockFocus**, – **lockFocusOn:**

useCacheWithDepth:

– (BOOL)**useCacheWithDepth:**(NXWindowDepth)*depth*

Creates a representation of type NXCachedImageRep and adds it to the NXImage's list of representations. Initially, the representation is nothing more than an empty area equal to the size of the image in an off-screen window with the specified *depth*. You must focus on the representation and draw the image. The following code shows how an NXImage might be created with the same appearance as a View.

```
id myImage;
NXRect theRect;

[myView getBounds:&theRect];
myImage = [[NXImage alloc] initSize:&frameRect.size];
[myImage useCacheWithDepth:NX_DefaultDepth];
if ( [myImage lockFocus] ) {
    [myView drawSelf:&theRect :1];
    [myImage unlockFocus];
}
```

depth should be one of the following enumerated values:

NX_DefaultDepth
NX_TwoBitGrayDepth
NX_EightBitGrayDepth
NX_TwelveBitRGBDepth
NX_TwentyFourBitRGBDepth

If successful in adding the representation, this method returns YES. If the size of the image has not been set or the cache can't be created for any other reason, it returns NO.

useDrawMethod:inObject:

– (BOOL)**useDrawMethod:**(SEL)*aSelector* **inObject:***anObject*

Creates a representation of type NXCustomImageRep and adds it to the NXImage object's list of representations. *aSelector* should name a method that can draw the image in the NXImage object's coordinate system, and that takes a single argument, an NXCustomImageRep. *anObject* should be the object that can perform the method.

This type of representation allows you to delegate responsibility for creating an image to another object within the program.

After invoking this method, you may need to explicitly set features of the newly created NXCustomImageRep, such as size, number of colors, and so on. This is true in particular if the NXImage has multiple image representations to choose from. A list of methods used to complete initialization is found in the class specification for NXCustomImageRep. Use NXImage's **lastRepresentation** method to access the newly created representation.

This method returns YES if it's successful in creating the representation, and NO if it's not.

useFromFile:

– (BOOL)**useFromFile:**(const char *)*filename*

Adds *filename* to the list of data sources used by the receiving NXImage. This method initializes lazily: the NXImage doesn't actually open *filename* or create an image representation from its data until an application attempts to composite or requests information about the NXImage. (Use the method **loadFromFile:** to immediately create an image representation for the data in a file.)

filename may be a full or relative pathname, and should include an extension that identifies the data type in the file. The mechanism that actually creates the image representation for *filename* will look for an NXImageRep subclass that handles that data type from among those registered with NXImage. By default, the files handled are those with the extensions "tiff", "tif", and "eps".

If a representation can be added to the NXImage, this method returns YES. If not, it returns NO. In the current implementation, it may return YES even if the *filename* file doesn't exist or contains bad data. To catch such cases, your application should check the value returned by the **lockFocus** methods or implement the delegate method **imageDidNotDraw:inRect:**. If *filename* contains TIFF or EPS data specifying more than one image, a separate representation is added for each one.

See also: – **initFromFile:**, + **registerImageRep:**, + **imageUnfilteredFileTypes** (NXImageRep)

useFromSection:

– (BOOL)**useFromSection:**(const char *)*name*

Adds *name* to the list of data sources used by the receiving NXImage. This method initializes lazily: the NXImage doesn't actually create a representation for the data in *name* until an application attempts to composite or requests information about the NXImage. When it does create a representation for *name*, the NXImage will look for data in the application's bundle.

For historical reasons, the mechanism that actually creates the image representation first looks for *name* in the application's executable file. It looks for a section with TIFF data in the __TIFF segment if *name* includes a ".tiff" extension, or for a section containing EPS data in the __EPS segment if *name* includes a ".eps" extension. If *name* has neither extension, both segments are searched, first after adding the appropriate extension to *name*, then for *name* alone, without an extension. If it finds sections in both segments, it creates both EPS and TIFF representations of the image.

Next, the mechanism searches for *name* files in the **lproj** directories in the application's main bundle. It searches for all file types (extensions) handled by all registered NXImageReps; by default, the files searched for include those with the extension "tiff", "tif", and "eps". It searches the language directories that the user specified for this application, or (if none) those specified by the user's default language preferences (see Application's **systemLanguages** method). If a file *name* is found with no extension, the mechanism opens the file and looks for a registered NXImageRep subclass that can handle the data (using the NXImage's **imageRepForStream:** class method).

If a section contains EPS or TIFF data for more than one version of the image, a representation will be created and added to the NXImage for each image specified. If an application bundle contains more than one file named *name* (each with a different extension), a representation will be created and added to the NXImage for each file whose data type is supported by a registered subclass of NXImageRep. The size of the NXImage is set from information taken from the TIFF fields or the EPS bounding box comment.

This method returns YES if a representation can be added to the NXImage, and NO if not. In the current implementation, it may return YES even if the section matching *name* contains bad data or no such section can be found.

See also: – **initFromSection:**

useRepresentation:

– (BOOL)**useRepresentation:**(NXImageRep *)*imageRep*

Adds *imageRep* to the receiving NXImage object's list of representations. If successful in adding the representation, this method returns YES. If not, it returns NO.

Any representation that's added by this method will belong to the NXImage and will be freed when the NXImage is freed. Representations can't be shared among NXImages.

After invoking this method, you may need to explicitly set features of the new representation, such as size, number of colors, and so on. This is true in particular if the NXImage has multiple image representations to choose from. A list of methods used to complete initialization is found in the class description for NXCustomImageRep.

See also: – **representationList**

write:

– **write:**(NXTypedStream *)*stream*

Writes the NXImage and all its representations to the typed stream *stream*. Representations created with **useFromFile:** or **useFromStream:** methods archive only the name of the data source. Representations created with the **loadFromFile:** or **loadFromStream:** methods or added via **useRepresentation:** method will archive the data for the image.

See also: – **read:**

writeTIFF:

– **writeTIFF:**(NXStream *)*stream*

Writes TIFF data for the representation that best matches the display device with the deepest frame buffer to *stream*. This method is a shorthand for **writeTIFF:allRepresentations:** with a *flag* of NO. Returns **self**.

writeTIFF:allRepresentations:

– **writeTIFF:**(NXStream *)*stream* **allRepresentations:**(BOOL)*flag*

Writes TIFF data for the representations to *stream*. If *flag* is YES, data will be written for each of the representations. If *flag* is NO, data will be written only for the representation that best matches the display device with the deepest frame buffer. Returns **self**.

If *stream* is positioned anywhere but at the beginning of the stream, this method will append the representation(s) it writes to the TIFF data it assumes is already in the stream. To do this, it must be able to read the TIFF header from the stream. Therefore, the stream must be opened for NX_READWRITE permission and must be seekable.

writeTIFF:allRepresentations:usingCompression:andFactor:

– **writeTIFF:**(NXStream *)*stream*
allRepresentations:(BOOL)*flag*
usingCompression:(int)*compressionType*
andFactor:(float)*compressionFactor*

Writes TIFF data for the representations to *stream*. If *flag* is YES, data will be written for each of the representations. If *flag* is NO, data will be written only for the representation that best matches the display device with the deepest frame buffer. The compression arguments let you specify a type of compression and the compression amount. The

compression types are listed in the section "Types and Constants." *compressionFactor* provides a hint for those compression types that implement variable comression ratios; currently only JPEG compression uses *compressionFactor.* Returns **self**.

Method Implemented By The Delegate

imageDidNotDraw:inRect:

– (NXImage *)**imageDidNotDraw:***sender* **inRect:**(const NXRect *)*aRect*

Implemented by the delegate to respond to a message sent by the *sender* NXImage when the *sender* was unable, for whatever reason, to composite or lock focus on its image. The delegate can return another NXImage to draw in the *sender*'s place. If not, it should return **nil** to indicate that *sender* should give up the attempt at drawing the image.

NXImageRep

Inherits From: Object

Declared In: appkit/NXImageRep.h

Class Description

NXImageRep is an abstract superclass; each of its subclasses knows how to draw an image from a particular kind of source data. While an NXImageRep subclass can be used directly, it's typically used through an NXImage object. An NXImage manages a group of representations, choosing the best one for the current output device.

There are four subclasses defined in the Application Kit:

Subclass	Source Data
NXBitmapImageRep	Tag Image File Format (TIFF) and other bitmap data
NXEPSImageRep	Encapsulated PostScript code (EPS)
NXCustomImageRep	A delegated method that can draw the image
NXCachedImageRep	A rendered image, usually in an off-screen window

Another subclass, N3DRIBImageRep, is defined in the 3D Graphics Kit. It renders an image from RenderMan Interface Bytestream (RIB) data. In applications that use the 3D Kit, NXImage will automatically use N3DRIBImageRep to handle RIB data.

You can define other NXImageRep subclasses for objects that render images from other types of source information. You make a subclass known to NXImage by invoking its **registerImageRep:** class method. The NXImageRep subclass informs NXImage of the data types it can support through its **imageUnfilteredFileTypes**, **imageUnfilteredPasteboardTypes**, and **canLoadFromStream:** class methods. Once an NXImageRep subclass is registered with NXImage, an instance of that subclass is created anytime NXImage encounters the type of data handled by that subclass.

Instance Variables

NXSize **size**;

size	The size of the image in screen pixels.

Method Types

Initializing	– initFromPasteboard:
Checking data types	+ canInitFromPasteboard: + canLoadFromStream: + imageFileTypes + imagePasteboardTypes + imageUnfilteredFileTypes + imageUnfilteredPasteboardTypes
Setting the size of the image	– setSize: – getSize:
Representation attributes	– setNumColors: – numColors – setAlpha: – hasAlpha – setBitsPerSample: – bitsPerSample – setPixelsHigh: – pixelsHigh – setPixelsWide: – pixelsWide – setOpaque: – isOpaque
Drawing the image	– draw – drawAt: – drawIn:
Archiving	– read: – write:

Class Methods

canInitFromPasteboard:

+ (BOOL)**canInitFromPasteboard:**(Pasteboard *)*pasteboard*

Returns YES if NXImageRep can handle the data represented by pasteboard. By default, this method returns NO.

This method invokes the **imageUnfilteredPasteboardTypes** class method and checks the list of types returned by that method against the data types in pasteboard. If it finds a match, it returns YES. When creating a subclass of NXImageRep that accepts image data from a

nondefault pasteboard type, you override the **imageUnfilteredPasteboardTypes** method to assure that this method returns the correct response.

See also: + **imageUnfilteredPasteboardTypes**

canLoadFromStream:

+ (BOOL)**canLoadFromStream:**(NXStream *)*stream*

Tests whether the receiving class can initialize an instance of itself from *stream*. Returns NO by default.

Override this method when you create a subclass of NXImageRep that can initialize itself from data in an NXStream. Your method should be able to seek in *stream* to determine whether it contains valid data for creating an instance of the class. It should return YES if there's a good chance that *stream* contains data of the type handled by the NXImageRep subclass. It should return NO only if it is clear from examining *stream* that there is no data that the NXImageRep subclass can handle. The determination should be able to be made by looking at the first few characters in *stream*. Before returning, your method should return the stream pointer to its initial position.

imageFileTypes

+ (const char *const *)**imageFileTypes**

Returns a null-terminated array of strings representing all file types supported by NXImageRep. The list includes both those types returned by the **imageUnfilteredFileTypes** class method and those that can be converted to a supported type by a user-installed filter service. Don't override this method when subclassing NXImageRep: it always returns a valid list for a subclass of NXImageRep that correctly overrides the **imageUnfilteredFileTypes** method.

By default, the returned array is empty.

See also: + **imageUnfilteredFileTypes**

imagePasteboardTypes

+ (const NXAtom *)**imagePasteboardTypes**

Returns a null-terminated array representing all pasteboard types supported by NXImageRep or one of its subclasses. The list includes both those types returned by the **imageUnfilteredPasteboardTypes** class method and those that can be converted by a user-installed filter service to a supported type. Don't override this method when

subclassing NXImageRep; it always return a valid list for a subclass of NXImageRep that correctly overrides the **imageUnfilteredPasteboardTypes** method.

By default, the returned array is empty.

See also: + **imageUnfilteredPasteboardTypes**

imageUnfilteredFileTypes

+ (const char *const *)**imageUnfilteredFileTypes**

Returns a null-terminated array of strings representing all file types (extensions) supported by the NXImageRep. By default, the returned array is emtpy.

When creating a subclass of NXImageRep, override this method to return a list of strings representing the file types supported. For example, NXBitmapImageRep implements the following code for this method:

```
+ (const char *const *)imageUnfilteredFileTypes
{
   static const char *const types[] = {"tiff", "tif", NULL};
   return types;
}
```

If your subclass supports the types supported by its superclass, you must explicitly get the array of types from the superclass and put them in the array returned by this method.

See also: + **imageFileTypes**, + **imageRepForFileType:** (NXImage)

imageUnfilteredPasteboardTypes

+ (const NXAtom *)**imageUnfilteredPasteboardTypes**

Returns a null-terminated array representing all pasteboard types supported by the NXImageRep. By default, the returned array is empty.

When creating a subclass of NXImageRep, override this method to return a list representing the pasteboard types supported. For example, NXBitmapImageRep implements the following code for this method:

```
+ (const NXAtom *)imageUnfilteredPasteboardTypes
{
   static NXAtom tiffTypes[2]={0, NULL};
   if (!tiffTypes[0]) tiffTypes[0] = NXTIFFPboardType;
   return tiffTypes;
}
```

If your subclass supports the types supported by its superclass, you must explicitly get the list of types from the superclass and add them to the array returned by this method.

See also: + **imagePasteboardTypes**, + **imageRepForPasteboardType:** (NXImage)

Instance Methods

bitsPerSample

– (int)**bitsPerSample**

Returns the number of bits used to specify a single pixel in each component of the data. If the image isn't specified by pixel values, but is device-independent, the return value will be NX_MATCHESDEVICE.

See also: – **setBitsPerSample:**

draw

– (BOOL)**draw**

Implemented by subclasses to draw the image at location (0.0, 0.0) in the current coordinate system. Subclass methods return YES if the image is successfully drawn, and NO if it isn't. This version of the method simply returns YES.

See also: – **drawAt:**, – **drawIn:**

drawAt:

– (BOOL)**drawAt:**(const NXPoint *)*point*

Translates the current coordinate system to the location specified by *point* and has the receiver's **draw** method draw the image at that point.

This method returns NO without translating or drawing if the size of the image has not been set. Otherwise, it returns the value returned by the **draw** method, which indicates whether the image is successfully drawn.

The coordinate system is not restored after it has been translated.

See also: – **draw**, – **drawIn:**

drawIn:

– (BOOL)**drawIn:**(const NXRect *)*rect*

Draws the image so that it fits inside the rectangle referred to by *rect*. The current coordinate system is first translated to the point specified in the rectangle and is then scaled so the image will fit within the rectangle. The receiver's **draw** method is then invoked to draw the image.

This method returns NO without translating, scaling, or drawing if the size of the image has not been set. Otherwise it returns the value returned by the **draw** method, which indicates whether the image is successfully drawn.

The previous coordinate system is not restored after it has been altered.

See also: – **draw**, – **drawAt:**

getSize:

– **getSize:**(NXSize *)*theSize*

Copies the size of the image to the structure referred to by *theSize*, and returns **self**. The size is provided in units of the base coordinate system.

See also: – **setSize:**

hasAlpha

– (BOOL)**hasAlpha**

Returns YES if the receiver has been informed that the image has a coverage component (alpha), and NO if not.

See also: – **setAlpha:**

initFromPasteboard:

– **initFromPasteboard:**(Pasteboard *)*pasteboard*

Initializes the NXImageRep with data from the given pasteboard.

isOpaque

– (BOOL)**isOpaque**

Returns YES if the receiver is opaque; NO otherwise. Use this method to test whether an NXImageRep completely covers the area within the rectangle returned by **getSize:**. Use the method **setOpaque:** to set the value returned by this method.

See also: **– hasAlpha**, **– getSize:**, **– setOpaque:**

numColors

– (int)**numColors**

Returns the number of color components in the image. For example, the return value will be 4 for images specified by cyan, magenta, yellow, and black (CMYK) or any other four components. It will be 3 for images specified by red, green, and blue (RGB), hue, saturation, and brightness (HSB), or any other three components. And it will be 1 for images that use only a gray scale. NX_MATCHESDEVICE is a meaningful return value for representations that vary their drawing depending on the output device.

See also: **– setNumColors:**

pixelsHigh

– (int)**pixelsHigh**

Returns the height of the image in pixels, as specified in the image data. If the image isn't specified by pixel values, but is device-independent, the return value will be NX_MATCHESDEVICE.

See also: **– setPixelsHigh:**

pixelsWide

– (int)**pixelsWide**

Returns the width of the image in pixels, as specified in the image data. If the image isn't specified by pixel values, but is device-independent, the return value will be NX_MATCHESDEVICE.

See also: **– setPixelsWide:**

read:

– **read:**(NXTypedStream *)*stream*

Reads the NXImageRep from the typed stream *stream*.

See also: – **write:**

setAlpha:

– **setAlpha:**(BOOL)*flag*

Informs the NXImageRep whether the image has an alpha component. *flag* should be YES if it does, and NO if it doesn't. Returns **self**.

See also: – **hasAlpha**

setBitsPerSample:

– **setBitsPerSample:**(int)*anInt*

Informs the NXImageRep that the image has *anInt* bits of data for each pixel in each component. If the image isn't specified by pixel values, but is device-independent, *anInt* should be NX_MATCHESDEVICE. Returns **self**.

See also: – **bitsPerSample**

setNumColors:

– **setNumColors:**(int)*anInt*

Informs the NXImageRep that the image has *anInt* number of color components. For color images with cyan, magenta, yellow, and black (CMYK) components, *anInt* should be 4, for color images with red, green, and blue (RGB) components, it should be 3, and for images that use only a gray scale, it should be 1. The alpha component is not included. NX_MATCHESDEVICE could be a meaningful value, if the representation varies its drawing depending on the output device. Returns **self**.

See also: – **numColors**

setOpaque:

– **setOpaque:**(BOOL)*flag*

Sets opacity of the NXImageRep's image.

setPixelsHigh:

– **setPixelsHigh:**(int)*anInt*

Informs the NXImageRep that the data specifies an image *anInt* pixels high. If the image isn't specified by pixel values, but is device-independent, *anInt* should be NX_MATCHESDEVICE. Returns **self**.

See also: – **pixelsHigh**

setPixelsWide:

– **setPixelsWide:**(int)*anInt*

Informs the NXImageRep that the data specifies an image *anInt* pixels wide. If the image isn't specified by pixel values, but is device-independent, *anInt* should be NX_MATCHESDEVICE. Returns **self**.

See also: – **pixelsWide**

setSize:

– **setSize:**(const NXSize *)*aSize*

Sets the size of the image in units of the base coordinate system, and returns **self**. This determines the size of the image when it's rendered; it's not necessarily the same as the width and height of the image in pixels as specified in the image data.

See also: – **getSize:**

write:

– **write:**(NXTypedStream *)*stream*

Writes the NXImageRep to the typed stream *stream*.

See also: – **read:**

NXJournaler

Inherits From: Object

Declared In: appkit/NXJournaler.h

Class Description

The NXJournaler class defines an object that lets an application record and play back events and sounds, a process called *journaling*. By using an NXJournaler object, an application can journal events flowing to one or more applications—including itself. Optionally, sound can be recorded synchronously with the events. Later, the recorded events and sound can be played back, reenacting the activities as they occurred during the recording. With journaling, you can implement event-based macros or complete self-running demonstrations for your application.

Journaling is initiated by creating a new NXJournaler object and sending it a **setEventStatus:soundStatus:eventStream:soundfile:** message. The status arguments may have the values NX_STOPPED, NX_PLAYING, and NX_RECORDING. The event stream argument is a stream to record to or play back from. If you're recording, any data in the stream will be overwritten. It's not currently possible to add to the end of an existing event stream. The sound file argument is the name of a sound file to record to or play back from.

When recording, by default all events going to any application are captured. Sometimes, you may not want certain applications to be recorded. For example, you might want to prevent the application that's recording the journal from being recorded. There are two ways to control this: with the defaults system and by sending a **setJournalable:** message to the Application object. Of the two, the defaults system is the more general.

To use the defaults system to control journaling, add this code to the **initialize** method of the object that will be controlling the journaling:

```
+ initialize
{
    static NXDefaultsVector myDefaults = {
        {"NXAllowJournaling", "NO"},
        {NULL}};
    NXRegisterDefaults([NXApp appName], myDefaults);
    return self;
}
```

This will prevent the application that contains the object from being journaled unless a user overrides the default for that application in the user's default database.

A user can also disallow journaling of any given application by adding an entry to the defaults database for that application. This would be done by entering the following command line in a Terminal window:

```
dwrite applicationName NXAllowJournaling NO
```

A less common way of allowing or disallowing journaling in an application is to send a **setJournalable:** message to the Application object. This allows more precise run-time control over journaling in that application.

Event recording may be aborted by clicking the right mouse button while holding down the Alternate key. (Note: For this to work, you must have the right mouse button enabled in the Preferences application.) Event playback can be aborted by typing a character with any key on the keyboard.

Instance Variables

None declared in this class.

Method Types

Initializing and freeing an NXJournaler	– init
	– free
Controlling journaling	– setEventStatus:soundStatus:eventStream:soundfile:
	– getEventStatus:soundStatus:eventStream:soundfile:
	– setRecordDevice:
	– recordDevice
Identifying associated objects	– speaker
	– listener
	– setDelegate:
	– delegate

Instance Methods

delegate

– **delegate**

Returns the NXJournaler's delegate.

See also: – **setDelegate:**

free

– **free**

Frees the NXJournaler. Send this message to an NXJournaler after you're completely done with it.

getEventStatus:soundStatus:eventStream:soundfile:

– **getEventStatus:**(int *)*eventStatusPtr*
soundStatus:(int *)*soundStatusPtr*
eventStream:(NXStream **) *streamPtr*
soundfile:(char **)*soundfilePtr*

Provides status information about the NXJournaler. Values returned at *eventStatusPtr* and *soundStatusPtr* can be NX_PLAYING, NX_RECORDING, or NX_STOPPED. *streamPtr* is the address of a pointer to the event stream. *soundfilePtr* is the address of a pointer to the name of the sound file. Any of the arguments may be NULL if you don't want that piece of information. Returns **self**.

See also: – **setEventStatus:soundStatus:eventStream:soundfile:**

init

– **init**

Initializes a newly allocated NXJournaler object. The delegate of the new object is **nil**. This is the designated initializer for an NXJournaler object. Returns **self**.

listener

– **listener**

Returns the listener used by the NXJournaler to communicate with other applications.

See also: – **speaker**

recordDevice

– (int)**recordDevice**

Returns whether sound is recorded from the CODEC microphone or from the DSP. The return value is either NX_CODEC or NX_DSP.

See also: – **setRecordDevice:**

setDelegate:

– **setDelegate:***anObject*

Sets the delegate used by the NXJournaler. The delegate is sent the method **journalerDidEnd:** when either playing or recording the journal finishes. If the journal was aborted, the delegate will first receive the message **journalerDidUserAbort:**. Returns **self**.

See also: – **delegate**

setEventStatus:soundStatus:eventStream:soundfile:

– **setEventStatus:**(int)*eventStatus*
soundStatus:(int)*soundStatus*
eventStream:(NXStream *)*stream*
soundfile:(const char *)*soundfile*

Controls the recording and playback of events and sounds. This is the main control point of the NXJournaler. The arguments *eventStatus* and *soundStatus* may be independently set to NX_STOPPED, NX_PLAYING, NX_RECORDING. By setting *eventStatus* to NX_RECORDING and *soundStatus* to NX_STOPPED, it's possible to record events without the sound. By setting *eventStatus* to NX_PLAYING and *soundStatus* to NX_RECORDING, it's possible to dub new sound over an existing event track.

The *stream* argument is the stream to record events to or playback events from. When recording, any preexisting data in the stream will be overwritten. It's not currently possible to record onto the end of an existing event stream.

The *soundfile* argument is the name of the file to record sound to or playback sound from.

If you logically OR NX_NONABORTABLEMASK into *eventStatus*, journaling will be made nonabortable.

See also: – getEventStatus:soundStatus:eventStream:soundfile:

setRecordDevice:

– setRecordDevice:(int)*device*

Sets whether sound is recorded from the CODEC microphone (the default device) or from the DSP. The constants NX_CODEC and NX_DSP can be used to specify the device. The recording from the DSP assumes that a peripheral is sending CD-quality data (stereo,16-bit linear, 44.1 kHz) to the DSP port. However, to save space, the data is reduced to a 22.05-kHz, mono sound.

See also: – recordDevice

speaker

– speaker

Returns the speaker used by the NXJournaler to communicate with the other applications.

See also: – listener

Methods Implemented By The Delegate

journalerDidEnd:

– journalerDidEnd:*journaler*

Responds to a message informing the delegate that recording or playback of the journal is finished or has been aborted.

See also: – journalerDidUserAbort:

journalerDidUserAbort:

– **journalerDidUserAbort:***journaler*

Responds to a message informing the delegate that the user has aborted the recording or playback session. A **journalerDidUserAbort:** message is sent when the NXJournaler in the controlling application receives notice from one of the controlled applications that the user has generated an abort event during recording or playback. The delegate receives this message just before the NXJournaler stops the recording or playback.

See also: – **journalerDidEnd:**

NXPrinter

Inherits From: Object

Declared In: appkit/NXPrinter.h

Class Description

An NXPrinter describes the printing capabilities of a particular make or type of printer, such as whether the printer can print in color, or whether it provides a particular font. In addition, some NXPrinters represent actual printer devices that are available to the computer for printing.

There are two ways to create an NXPrinter:

- If you want an abstract object that doesn't represent an actual printer but gives the printing attributes of a type of printer, you use the **newForType:** class method, passing a printer type (a string) as the argument. A list of printer types that are recognized by the computer is available through the **printerTypes:custom:** class method.
- To find or create an NXPrinter that corresponds to an actual printer device, you use one of the **newForName:** class methods, passing, at least, the name of a printer. A list of printer names can be retrieved through the **prdb_get()** function.

Once you've gotten an NXPrinter, there's only one thing you can do with it: Retrieve information regarding the object's type or regarding the actual printer that the object represents (if it represents an actual printer). You can't change the information in an NXPrinter, nor can you use an NXPrinter object to initiate or control a printing job. In addition, NXPrinter instances are owned by the NXPrinter class; you never free them directly.

Printer types are described in files written in the PostScript Printer Description (PPD) format. The printer types that NeXT provides are in localized subdirectories of **/NextLibrary/PrinterTypes**. When you create an NXPrinter object for a particular type, the object reads the corresponding PPD file, manipulates the information it finds there, and stores the data in named tables. Commonly needed items, such as whether a printer is color or the size of the page on which it prints, are available directly through methods defined by NXPrinter (methods such as **isColor** and **pageSizeForPaper:**). Any bit of information in the PPD tables can be retrieved through more general methods such as **stringForKey: inTable:**, as explained below.

Note: To understand what the NXPrinter tables contain, you need to be acquainted with the PPD file format. This is described in *PostScript Printer Description File Format Specification, version 4.0*, available from Adobe Systems Incorporated. The rest of this class description assumes a familiarity with the concepts and terminology presented in the Adobe manual. A brief summary of the PPD format is given below; PPD terms defined in the Adobe manual are shown in italic.

PPD Format

A PPD file statement, or *entry*, associates a *value* with a *main keyword*:

**mainKeyword: value*

The asterisk is literal; it indicates the beginning of a new entry.

For example:

```
*ModelName: "MMimeo Machine"
*3dDevice: False
```

A main keyword can be qualified by an *option keyword*:

**mainKeyword optionKeyword: value*

For example:

```
*PaperDensity Letter: "0.1"
*PaperDensity Legal: "0.2"
*PaperDensity A4: "0.3"
*PaperDensity B5: "0.4"
```

In addition, any number of entries may have the same main keyword with no option keyword yet give different values:

```
*InkName: ProcessBlack/Process Black
*InkName: CustomColor/Custom Color
*InkName: ProcessCyan/Process Cyan
*InkName: ProcessMagenta/Process Magenta
*InkName: ProcessYellow/Process Yellow
```

Option keywords and values can sport *translation strings*. A translation string is a textual description, appropriate for display in a user interface, of the option or value. An option or value is separated from its translation string by a slash:

```
*Resolution 300dpi/300 dpi: " ... "
*InkName: ProcessBlack/Process Black
```

In the first example, the **300dpi** option would be presented in a user interface as "300 dpi." The second example assigns the string "Process Black" as the translation string for the **ProcessBlack** value.

Entries that have an ***OrderDependency** or ***UIConstraints** main keyword are treated specially by NXPrinter. Such entries take the following forms (the bracketed elements are optional):

*OrderDependency: *real section mainKeyword* [*optionKeyword*]
*UIConstraint: *mainKeyword1* [*optionKeyword1*] *mainKeyword2* [*optionKeyword2*]

There may be more than one UIConstraints entry with the same *mainKeyword1* or *mainKeyword1/optionKeyword1* value. Below are some examples of ***OrderDependency** and ***UIConstraints** entries:

```
*OrderDependency: 10 AnySetup *Resolution
*UIConstraint: *Option3 None *PageSize Legal
*UIConstraint: *Option3 None *PageRegion Legal
```

Explaining these entries is beyond the scope of this documentation; however, it's important to note their forms in order to understand how they're represented in the NXPrinter tables.

NXPrinter Tables

NXPrinter defines a five key-value tables to store PPD information. The tables are identified by string names, as given below:

String Name	Contents
PPD	General information about a printer type. The table contains the values for all entries in a PPD file except those with ***OrderDependency** and ***UIConstraints** main keywords. The values in this table don't include the translation strings.
PPDOptionTranslation	Option keyword translation strings.
PPDArgumentTranslation	Value translation strings.
PPDOrderDependency	***OrderDependency** values.
PPDUIConstraint	***UIConstraints** values.

There are two principle methods for retrieving data from the NXPrinter tables:

- **stringForKey:inTable:** returns the value for the first occurence of a given key in the given table.
- **stringListForKey:inTable:** returns an array of values, one for each occurence of the key.

For both methods, the first argument is a string that names a key—which part of a PPD file entry the key corresponds to depends on the table (as explained in the following sections). The second argument names the table that you want to look in. The values that are returned by these methods, whether singular or in an array, are always strings, even if the value wasn't a quoted string in the PPD file.

The NXPrinter tables store data as ASCII text, thus the two methods described above are sufficient for retrieving any value from any table. NXPrinter provides a number of other methods, such as **booleanForKey:inTable:** and **intForKey:inTable:**, that retrieve single values and coerce them, if possible, into particular data types. The coercion doesn't affect the data that's stored in the table (it remains in ASCII format).

To check the integrity of a table, use the **isKey:forTable:** and **statusForTable:** methods. The former returns a boolean that indicates whether the given key is valid for the given table; the latter returns an error code that describes the general state of a table (in particular, whether it actually exists).

Retrieving Values from the PPD Table

Keys for the PPD table are strings that name a main keyword or main keyword/option keyword pairing (formatted as "*mainKeyword/optionKeyword*"). In both cases, you exclude the main keyword asterisk. The following example creates an NXPrinter and invokes **stringForKey:inTable:** to retrieve the value for an un-optioned main keyword:

```
/* Create an NXPrinter object for a printer type. */
NXPrinter *prType = [NXPrinter
                     newForType:"My_Mimeo_Machine"]

char *sValue = [prType stringForKey:"3dDevice" inTable:"PPD"];
/* sValue is "False". */
```

To retrieve the value for a main/option pair, you pass the keywords formatted as "*mainKeyword/optionKeyword*":

```
char *sValue = [prType stringForKey:"PaperDensity/A4"
                           inTable:"PPD"];
/* sValue is "0.3". */
```

You can use **stringForKey:inTable:** to determine if a main keyword has options. If you pass a main keyword (only) as the first argument to the method, and if that keyword has options in the PPD file, the method returns NULL. If it doesn't have options, it returns the value of the first occurence of the main keyword:

```
char *sValue = [prType stringForKey:"PaperDensity" inTable:"PPD"];
/* sList is NULL */

char *sValue = [prType stringForKey:"InkName" inTable:"PPD"];
/* sList is "ProcessBlack" */
```

To retrieve the values for all occurrences of an un-optioned main keyword, use the **stringListForKey:inTable:** method:

```
char **sList = [prType stringListForKey:"InkName" inTable:"PPD"];
/* sList[0] is "ProcessBlack",
   sList[1] is "CustomColor",
   sList[2] is "ProcessCyan", and so on. */
```

In addition, **stringListForKey:inTable:** can be used to retrieve all the options for a main keyword (given that the main keyword has options):

```
char **sList = [prType stringListForKey:"PaperDensity"
                              inTable:"PPD"];
/* sList[0] is "Letter",
   sList[1] is "Legal",
   sList[2] is "A4", and so on. */
```

Retrieving Values from the Option and Argument Translation Tables

A key to a translation table is like that to the PPD table: It's a main keyword or main/option keyword pair (again excluding the asterisk). However, the values that are returned from the translation tables are the translation strings for the option or argument (value) portions of the PPD file entry. For example:

```
char *sValue = [prType stringForKey:"Resolution/300dpi"
                              inTable:"PPDOptionTranslation"];
/* sValue is "300 dpi". */

char **sList = [prType stringListForKey:"InkName"
                              inTable:"PPDArgumentTranslation"];
/* sList[0] is "Process Black",
   sList[1] is "Custom Color",
   sList[2] is "Process Cyan", and so on. */
```

As with the PPD table, requesting an array of strings for an un-optioned main keyword returns the keyword's options (if it has any).

Retrieving Values from the Order Dependency Table

As mentioned earlier, an order dependency entry takes this form:

*orderDependency: *real section mainKeyword* [*optionKeyword*]

These entries are stored in the PPDOrderDependency table. To retrieve a value from this table, you always use **stringListForKey:inTable:**. The value that you pass as the key is, again, a main keyword or main/option pair; however, these values correspond to the *mainKeyword* and *optionKeyword* parts of an order dependency entry's value. As with the other tables, the main keyword's asterisk is excluded. The method returns an array of two strings that correspond to the *real* and *section* values for the entry. For example:

```
char **sList = [prType stringListForKey:"Resolution"
                              inTable:"PPDOrderDependency"]
/* sList[0] = "10", sList[1] = "AnySetup" */
```

Retrieving Values from the UIConstraints Table

Retrieving a value from the PPDUIConstraints table is similar to retrieving a value from the PPDOrderDependency table: You always use **stringListForKey:inTable:** and the key corresponds to elements in the entry's value. Given the following form (as described earlier), the key corresponds to *mainKeyword1/optionKeyword1*:

*UIConstraint: *mainKeyword1* [*optionKeyword1*] *mainKeyword2* [*optionKeyword2*]

The array that's returned by **stringListForKey:inTable:** contains the *mainKeyword2* and *optionKeyword2* values (with the keywords stored as separate elements in the array) for every ***UIConstraints** entry that has the given *mainKeyword1/optionKeyword1* value. For example:

```
char **sList = [prType stringListForKey:"Option3/None"
                              inTable:"PPDUIConstraints"]
/* sList[0] = "PageSize", sList[1] = "Legal",
   sList[2] = "PageRegion", sList[3] = "Legal" */
```

Note that the main keywords that are returned in the array don't have asterisks. Also, the array that's returned always alternates main and option keywords. If a particular main keyword doesn't have an option associated with it, the string for the option will be empty (but the entry in the array for the option *will* exist).

Instance Variables

const char ***printerName**;
const char ***hostName**;
const char ***domainName**;
const char ***printerType**;

printerName	The printer's name.
hostName	The name of the printer's host computer.
domainName	The name of the printer's domain.
printerType	The name of the printer's type.

Method Types

Finding an NXPrinter	+ newForName: + newForName:host: + newForName:host:domain:includeUnavailable: + newForType: + printerTypes:custom:
Printer attributes	– domain – host – name – note – type – isReallyAPrinter
Retrieving specific information	– acceptsBinary – imageRectForPaper: – pageSizeForPaper: – isColor – isFontAvailable: – isValid – languageLevel – isOutputStackInReverseOrder

Querying the NXPrinter tables	– booleanForKey:inTable: – dataForKey:inTable:length: – floatForKey:inTable: – intForKey:inTable: – rectForKey:inTable: – sizeForKey:inTable: – stringForKey:inTable: – stringListForKey:inTable: – statusForTable: – isKey:inTable:

Class Methods

newForName:

+ (NXPrinter *)**newForName:**(const char *)*name*

Returns the NXPrinter with the given name; returns the value returned by

```
[self newForName:name host:NULL domain:NULL includeUnavailable:NO]
```

See also: + newForName:host:domain:includeUnavailable:, + newForType:

newForName:host:

+ (NXPrinter *)**newForName:**(const char *)*name* **host:**(const char *)*hostName*

Returns the NXPrinter with the given name and host; returns the value returned by

```
[self newForName:name host:hostName domain:NULL
includeUnavailable:NO]
```

See also: + newForName:host:domain:includeUnavailable:, + newForType:

newForName:host:domain:includeUnavailable:

+ (NXPrinter *)**newForName:**(const char *)*name*
host:(const char *)*hostName*
domain:(const char *)*domain*
includeUnavailable:(BOOL)*includeFlag*

Returns an NXPrinter that represents an actual printer with the given name, host, and domain. If *hostName* or *domainName* is NULL, the first printer (with the given name) found on any host or domain is used. If *hostName* is an empty string, the local host is used. If *includeFlag* is NO, then the matching printer must be available for printing, otherwise **nil** is returned. If the flag is YES, the availability of the printer is ignored.

See also: + **newForType:**

newForType:

+ (NXPrinter *)**newForType:**(const char *)*type*

Returns an NXPrinter object that contains information for the given type; the object doesn't correspond to an actual printer. The *type* argument should be an element in the array returned by **printerTypes:custom:**.

See also: + **newForName:host:domain:includeUnavailable:**, + **printerTypes:custom:**

printerTypes:custom:

+ (char **)**printerTypes:**(BOOL)*normalFlag* **custom:**(BOOL)*customFlag*

Returns a pointer to an array of strings that give the names of the printer types that are recognized by the computer. The flag arguments indicate whether to include normal printer types, custom types, or both.

A printer type is represented by a PPD file (extension ".ppd"). This method searches for normal PPD files directly, or in bundles, in the following directories:

/NextLibrary/PrinterTypes
~/Library/PrinterTypes
/HostLibrary/PrinterTypes
/LocalLibrary/PrinterTypes

Custom PPD files are searched for in the "CustomPrinters" subdirectory (or bundles therein) in each of the above.

See also: + **newForType**

Instance Methods

acceptsBinary

– (BOOL)**acceptsBinary**

Returns YES if the NXPrinter accepts binary PostScript data. Otherwise returns NO.

booleanForKey:inTable:

– (BOOL)**booleanForKey:**(const char *)*key* **inTable:**(const char *)*table*

Returns a boolean value for the given key in the given table: YES is returned if the value, which is stored as ASCII text, is "YES", "TRUE", or names a non-negative integer. Otherwise, this returns NO. *key* should be formed as described in the class description, above.

See also: – **stringForKey:inTable:**

dataForKey:inTable:length:

– (void *)**dataForKey:**(const char *)*key*
inTable:(const char *)*table*
length:(int *)*bytes*

Returns a pointer to untyped data for the given key in the given table. The length of the data, in bytes, is returned by reference in *bytes*. *key* should be formed as described in the class description, above.

See also: – **stringForKey:inTable:**

domain

– (const char *)**domain**

Returns the name of the domain in which the NXPrinter's printer entry resides. If the object doesn't represent an actual printer, this returns a pointer to NULL.

See also: + **newForName:host:domain:includeUnavailable:**

floatForKey:inTable:

– (float)**floatForKey:**(const char *)*key* **inTable:**(const char *)*table*

Returns a floating-point value for the given key in the given table; returns 0.0 if the value, which is stored as ASCII text, can't be coerced to **float**. *key* should be formed as described in the class description, above.

See also: – **stringForKey:inTable:**

free

Never invoke this method. NXPrinter objects are owned by the NXPrinter class—you never free them yourself.

host

– (const char *)**host**

Returns the name of the host to which the printer is connected. If the object doesn't represent an actual printer, this returns a pointer to NULL.

See also: + **newForName:host:**, + **newForName:host:domain:includeUnavailable:**

imageRectForPaper:

– (NXRect)**imageRectForPaper:**(const char *)*paperType*

Returns the printing rectangle—the area of the page that's available for printing—for the named paper type. The selection of paper type names depends on the NXPrinter's type; typical names include "Legal", "Letter", "A4", and "B5".

See also: – **pageSizeForPaper:**

init

You never invoke this method. To create an NXPrinter, use one of the **newFor...** class methods.

intForKey:inTable:

– (int)**intForKey:**(const char *)*key* **inTable:**(const char *)*table*

Returns an integer value for the given key in the given table; returns 0 if the value, which is stored as ASCII text, can't be coerced to **int**. *key* should be formed as described in the class description, above.

See also: – **stringForKey:inTable:**

isColor

– (BOOL)**isColor**

Returns YES if the NXPrinter can print in color. Otherwise returns NO.

isFontAvailable:

– (BOOL)**isFontAvailable:**(const char *)*fontName*

Returns YES if the named font is available to the NXPrinter; otherwise returns NO. Font names are formed as in an invocation of Font's **useFont:** method; examples include "Helvetica-Bold", "Times-Roman", and "Courier-BoldOblique".

isKey:inTable:

– (BOOL)**isKey:**(const char *)*key* **inTable:**(const char *)*table*

Returns YES if *key* is a key to *table* (which must name one of the NXPrinter tables listed in the class description, above).

See also: – **statusForTable:**

isOutputStackInReverseOrder

– (BOOL)**isOutputStackInReverseOrder**

Returns YES if the printer outputs pages in reverse page order, otherwise returns NO. By being printed in reverse order, the pages in the resulting output stack will be in the correct (first-to-last) order (assuming that the printer produces pages face-up).

isReallyAPrinter

– (BOOL)**isReallyAPrinter**

Returns YES if the NXPrinter corresponds to an actual printer device. Otherwise returns NO.

isValid

– (BOOL)**isValid**

Returns YES if the NXPrinter is valid—if its internal state matches physical reality. Otherwise returns NO. This is important only for NXPrinters that correspond to actual printers.

See also: – **statusForTable:**

languageLevel

– (int)**languageLevel**

Returns 1 or 2 as the NXPrinter recognizes the PostScript Language Level I or Level II.

name

– (const char *)**name**

Returns the name of the NXPrinter. If the object doesn't represent an actual printer, this returns a pointer to NULL.

See also: + **newForName:**, + **newForName:host:domain:includeUnavailable:**

note

– (const char *)**note**

Returns the comment that's associated with the NXPrinter. If the object doesn't represent an actual printer, this returns a pointer to NULL. The text for the note is set through the PrintManager application.

pageSizeForPaper:

– (NXSize)**pageSizeForPaper:**(const char *)*paperType*

Returns the size of the page for the named paper type. The selection of paper type names depends on the NXPrinter's type; typical names include "Legal", "Letter", "A4", and "B5".

See also: – **imageRectForPaper:**

rectForKey:inTable:

– (NXRect)**rectForKey:**(const char *)*key* **inTable:**(const char *)*table*

Returns an NXRect for the given key in the given table. The individual fields are set to 0.0 if the value, which is stored as ASCII text, can't be fit into an NXRect structure. *key* should be formed as described in the class description, above.

See also: – **stringForKey:inTable:**

sizeForKey:inTable:

– (NXSize)**sizeForKey:**(const char *)*key* **inTable:**(const char *)*table*

Returns an NXSize for the given key in the given table. The individual fields are set to 0.0 if the value, which is stored as ASCII text, can't be fit into an NXSize structure. *key* should be formed as described in the class description, above.

See also: – **stringForKey:inTable:**

statusForTable:

– (int)**statusForTable:**(const char *)*table*

Returns one of the following constants to indicate the status of the given table:

Constant	Meaning
NX_PRINTERTABLEOK	The table is valid.
NX_PRINTERTABLENOTFOUND	The table doesn't exist in this NXPrinter.
NX_PRINTERTABLEERROR	The table exists but is invalid.

See also: – **isValid**

stringForKey:inTable:

– (const char *)**stringForKey:**(const char *)*key* **inTable:**(const char *)*table*

Returns a pointer to the ASCII text that corresponds to *key* in the given table. If the table contains more than one entry with this key, the value of the first entry is returned. A pointer to NULL is returned if the table doesn't contain a key that precisely matches *key*. See the class description, above, for more information on this method.

See also: – **stringListForKey:inTable:**

stringListForKey:inTable:

– (const char **)**stringListForKey:**(const char *)*key* **inTable:**(const char *)*table*

Returns a pointer to an array of strings; each string gives the ASCII text that corresponds to an entry that has the given key in the given table. If *key* names a main keyword for which there are (in the table) option keywords, the returned array contains the option keywords. See the class description, above, for more information on this method.

Note that it's the invoker's responsibility to free the array that's returned by this method, but not the contents of the array.

See also: – **stringForKey:inTable:**

type

– (const char *)**type**

Returns a string that names the NXPrinter's type.

See also: – **newForType:**, – **newForName:**

NXSelection

Inherits From: Object

Declared In: appkit/NXSelection.h

Class Description

The NXSelection class defines an object that describes a selection within a document. An NXSelection is an immutable description; it may be held by the system or other documents, and it cannot change over time.

Because a selection description can't be changed once it has been exported, it's a good idea to construct general descriptions that can survive changes to a document and don't require specific selection information to be stored in the document. This may be simple or complex, depending upon the application. For example, a painting application might describe a selection in an image as a simple rectangle. This description doesn't require that any information be stored in the image's file, and the description can be expected to remain valid through the life of the image. An object-based drawing application might describe a selection as a list of object identifiers (though *not* **id**s), where an object identifier is unique through the life of the document. Based on this list, a selection could be meaningfully reconstructed, even if new objects are added to the document or selected objects are deleted. Such a scheme doesn't require that any selection-specific information be stored in the document's file, with the benefit that links can be made to read-only documents.

Maintaining a character-range selection in a text document is more problematic. A possible solution is to insert a selection-begin and selection-end marker that refer to a specific selection into the text stream. A selection description would then refer to a specific selection marker. This solution requires that selection state information be stored and maintained within the document. Furthermore, this information generally shouldn't be purged from the document, because the document can't know how many references to the selection exist. (References to the selection could be stored with documents on removable media, like floppy disks.) This selection-state information should be maintained as long as it refers to any meaningful data. For this reason, it's desirable to describe selection in a manner that doesn't require that selection-state information be maintained in the document whenever possible.

Three well-known selection descriptions can apply to any document: the empty selection, the entire document, and the abstract concept of the current selection. NXSelection objects for these selections are returned by the **emptySelection**, **allSelection**, and **currentSelection** class methods.

Since an NXSelection may be used in a document that is read by machines with different architectures, care should be taken to write machine-independent descriptions. For example, using a binary structure as a selection description will fail on a machine where an identically defined structure has a different size or is kept in memory with different byte ordering. Exporting (and then parsing) ASCII descriptions is often a good solution. If binary descriptions must be used, it is prudent to preface the description with a token specifying the description's byte ordering.

It may also be prudent to version-stamp selection descriptions, so that old selections can be accurately read by updated versions of an application.

Instance Variables

None declared in this class.

Method Types

Returning special Selections	+ emptySelection + allSelection + currentSelection
Initializing a Selection	– initWithDescription:length: – initWithDescriptionNoCopy:length: – initFromPasteboard:
Copying a Selection	– copyFromZone:
Describing a Selection	– descriptionOfLength:
Comparing Selections	– isEqual: – isWellKnownSelection
Writing the Selection to the Pasteboard	– writeToPasteboard:

Class Methods

allSelection

+ allSelection

Returns the shared instance of the well-known selection representing an entire document.

currentSelection

+ currentSelection

Returns the shared instance of the well-known selection representing the abstract concept of the current selection. The current selection never describes a specific selection; it describes a selection that may change frequently.

emptySelection

+ emptySelection

Returns the shared instance of the well-known selection representing no data.

Instance Methods

copyFromZone:

– **copyFromZone:**(NXZone *)*zone*

If the receiving NXSelection is a shared instance of a well-known selection, returns the NXSelection. Otherwise, returns a copy of the NXSelection allocated from *zone*.

descriptionOfLength:

– (const void *)**descriptionOfLength:**(int *)*count*

Returns a pointer to the description of the selection and fills in the integer indicated by *count* with the length (in bytes) of the description. The description was set with one of the **initWithDescription...** methods when the selection was created.

initFromPasteboard:

– **initFromPasteboard:**(Pasteboard *)*pasteboard*

Initializes a newly allocated NXSelection instance from data on the specified pasteboard. If the NXSelection can't be initialized for any reason (for example, if data of type NXSelectionPboardType isn't found on the pasteboard) the new instance is freed and **nil** is returned.

See also: – **writeToPasteboard:**

initWithDescription:length:

– **initWithDescription:**(const void *)*description* **length:**(int)*count*

Initializes a newly allocated NXSelection instance using a copy of the data indicated by *description*, of length *count*, to describe the selection. If *count* is –1, *description* is assumed to be null-terminated data. *description* can be in any format and of any length, but should be architecture independent. It's a good idea to include version information in such a description.

initWithDescriptionNoCopy:length:

– **initWithDescriptionNoCopy:**(const void *)*description* **length:**(int)*count*

Initializes a newly allocated NXSelection instance using the data indicated by *description*, of length *count*, to describe the selection. The description will not be copied nor modified by the NXSelection, and it ought to be persistent and unchanging. If *count* is –1, *description* is assumed to be null-terminated data. *description* can be in any format and of any length, but should be architecture independent. It's a good idea to include version information in such a description.

isEqual:

– (BOOL)**isEqual:***otherSel*

Compares the receiving NXSelection with another NXSelection, specified by *otherSel*. Returns YES if they describe the same selection, and NO otherwise.

isWellKnownSelection

– (BOOL)**isWellKnownSelection**

Returns YES if the NXSelection is one of the well-known selection types, and NO otherwise. There are well-known selection types for an entire document, the current selection, and for an empty selection.

See also: + allSelection, + currentSelection, + emptySelection

writeToPasteboard:

– **writeToPasteboard:**(Pasteboard *)*pasteboard*

Writes the NXSelection to the pasteboard *pasteboard*. A copy of the selection can then be retrieved by initializing a new NXSelection from the pasteboard using **initFromPasteboard:**.

NXSpellChecker

Inherits From: Object

Declared In: appkit/NXSpellChecker.h

Class Description

The NXSpellChecker class gives any application an interface to the NeXT spell-checking service. To handle all its spell checking, an application needs only one instance of NXSpellChecker. It provides a panel in which the user can record decisions about words that are suspect. To check the spelling of a piece of text, the application:

- Includes in its user interface a menu item (or a button or command) by which the user will request spell checking and makes the text available by way of an object that adopts certain protocols.
- Creates an instance of the NXSpellChecker class and sends it a **checkSpelling:of:** message. The message's arguments identify the object to be checked and one of several modes of checking.

Typical code to make use of NXSpellChecker might be:

```
[[NXSpellChecker sharedInstance] checkSpelling:NX_CheckSpelling of:self]
```

The first argument of the **checkSpelling:of:** method defines the scope of checking: whether the request is just to count words or actually to search for misspellings, and whether the action applies to the entire text and where to start. The second argument is the object that provides the text to be checked.

The application may choose to split a document's text into segments and check them separately. This will be necessary when the text has segments in different languages. Spell checking is invoked for one language at a time, so a document that contains portions in three languages will require at least three checks.

The object that provides the text must adopt the following protocols (of which the first two are mandatory, while the others are required only if you want to take advantage of functions they provide):

NXReadOnlyTextStream	This is how the NXSpellChecker reads the text.
NXSelectText	This is how the NXSpellChecker highlights a misspelled word in the display.
NXChangeSpelling	A message in this protocol is sent down the responder chain when the user presses the Correct button.
NXIgnoreMisspelledWords	When the object being checked responds to this protocol, the spell server keeps a list of words that are acceptable in the document, and enables the Ignore button in the Spelling panel.

Dictionaries and Word Lists

The process of checking spelling makes use of three references:

- A dictionary registered with the system's spell-checking service. When the Spelling panel first appears, by default it shows the dictionary for the user's preferred language. The user may select a different dictionary from the list in the Spelling Panel.
- The user's "learn" list of correctly-spelled words in the current language. The NXSpellChecker updates the list when the user presses the Learn or Forget buttons in the Spelling panel.
- The document's list of words to be ignored while checking it. The NXSpellChecker updates its copy of this list when the user presses the Ignore button in the Spelling panel.

A word is misspelled if none of these three accepts it. When the spelling server finds such a word, it sends messages to the object being checked telling it to select the misspelled word and scroll the document's display to make the selection visible.

Matching a List of Ignored Words with the Document It Belongs To

Notice that "object providing text to be checked" isn't quite the same as "document." In the course of processing a document, an application might run several checks, based on different parts or different versions of the text. But they'd all belong to the same document. The NXSpellChecker keeps a separate "ignored words" list for each document that it checks. However, when it receives a **checkSpelling:of:** message, it doesn't know to what document the text belongs. To match "ignored words" lists to documents, each time the NXSpellChecker starts a search, it asks the application for a *spell client tag*. The tag is an arbitrary integer that lasts only while the application runs; it serves only to distinguish one document from the others being checked, and to match each "ignored words" list to a document.

When the application closes a document, it may choose to retrieve the "ignored words" list and save it along with the document. To get back the right list, it sends the NXSpellChecker an **ignoredWordsForSpellDocument:** message, using the document's tag to identify it. When the application has closed a document, it should notify the NXSpellChecker that the document's "ignored words" list can now be discarded by sending it a **closeSpellDocument:** message, again using the tag to identify the document that has closed.

Instance Variables

None declared in this class.

Method Types

Getting the NXSpellChecker	+ sharedInstance + sharedInstance:
Modifying the spelling panel	– spellingPanel – setAccessoryView: – accessoryView – setWordFieldValue:
Setting the language	– setLanguage – language:
Checking spelling	– checkSpelling:of: – checkSpelling:of:wordCount:
Managing ignored words	– setIgnoredWordsForSpellDocument: – ignoredWords:forSpellDocument: – closeSpellDocument:

Class Methods

sharedInstance

+ sharedInstance

Returns an instance of the NXSpellChecker class. If the application has not yet asked for an NXSpellChecker object, this method allocates and initializes a new instance. Invoking **sharedIntance** has the same effect as invoking **sharedInstance:** with *flag* set to YES.

See also: + sharedInstance:

sharedInstance:

+ **sharedInstance:**(BOOL)*flag*

Returns an instance of the NXSpellChecker class. If an instance is already in existence, this method simply returns it. When the application had not yet asked for an NXSpellChecker object, if *flag* is YES, this method allocates and initializes a new instance. When *flag* is NO, the method does not create a new instance and returns **nil**.

Instance Methods

accessoryView

– **accessoryView**

Returns the customized View of the Spelling panel previously established by **setAccessoryView:**.

See also: – **setAccessoryView:**

checkSpelling:of:

– (BOOL)**checkSpelling:**(NXSpellCheckMode)*mode*
of:(id <NXReadOnlyTextStream, NXSelectRange>)*anObject*

The first argument *mode* indicates the mode in which the spelling checker will operate, as defined in the enumeration set NXSpellCheckMode. Values that can be used with **checkSpelling:of:** are as follows:

NX_CheckSpelling	Checks spelling of the entire text stream, in sequence from the current character offset to the end, then from the start to the current character offset.
NX_CheckSpellingFromStart	Checks spelling of the entire text stream, from start to end.
NX_CheckSpellingToEnd	Checks spelling from the current character offset to the end of the text stream.
NX_CheckSpellingInSelection	Checks spelling in the selected portion of the text stream.

The method returns YES when the check has found a misspelled word, NO when it has searched the designated portion of the text without finding one.

See also: –**checkSpelling:of:wordCount:**, – **currentCharacterOffset** (NXReadOnlyTextStream protocol).

checkSpelling:of:wordCount:

– (BOOL)**checkSpelling:**(NXSpellCheckMode)*mode*
of:(id <NXReadOnlyTextStream, NXSelectRange>)*anObject*
wordCount:(int *)*theCount*

This method looks for a misspelled word and at the same time counts the number of words scanned. *mode* indicates the mode of search or count, a member of the enumeration type NXSpellCheckMode. Values that can be used with **checkSpelling:of:wordCount:** are as follows:

NX_CheckSpelling	Checks spelling and counts words for the entire text stream, in sequence from the current character offset to the end, then from the beginning to the current character offset.
NX_CheckSpellingFromStart	Checks spelling and counts words for the entire text stream, from start to end.
NX_CheckSpellingToEnd	Checks spelling and counts words from the current character offset to the end of the text stream.
NX_CheckSpellingInSelection	Checks spelling and counts words in the selected portion of the text stream.
NX_CountWords	Counts the number of words in the entire text stream; doesn't check spelling.
NX_CountWordsToEnd	Counts the number of words from the position reported by current character offset to the end of the text stream; doesn't check spelling.
NX_CountWordsInSelection	Counts the number of words in the selected portion of the text stream; doesn't check spelling.

theCount points to an integer where the receiver will put the number of words. If for some reason the spelling checker is unable to count words, it puts –1 there.

The method returns YES when the check has found a misspelled word, NO when it has searched the designated portion of the text without finding one.

See also: – **checkSpelling:of:**, – **currentCharacterOffset** (NXReadOnlyTextStream protocol)

closeSpellDocument:

– **closeSpellDocument:**(int)*tag*

Notifies the NXSpellChecker object that the user has finished with the document identified by *tag*. The NXSpellChecker can then discard the temporary list of ignorable words that it compiled for this document. The argument identifies the document by a numeric tag, previously returned by **spellDocumentTag**. Returns **self**.

See also: – **spellDocumentTag** (NXIgnoreMisspelledWords protocol)

ignoredWordsForSpellDocument:

– (char **)**ignoredWordsForSpellDocument:**(int)*tag*

Returns the list of ignored words that the spell checker has accumulated for the document identified by the argument *tag*. The application may want to preserve the list of ignored words so that they can be ignored in future checks of the same document. To preserve the ignored word list, it should invoke this method *after* spelling has been checked but *before* it sends a **closeSpellDocument:** message. The argument identifies the document by the tag that **spellDocumentTag** returned previously.

See also: – **spellDocumentTag** (NXIgnoreMisspelledWords protocol)

language

– (const char *)**language**

Returns the character string that identifies the English name of the currently selected language. If the application elects to temporarily override the current language (by invoking **setLanguage:**), this method will be useful to record the current language so that it can subsequently be restored. Otherwise, the application will not ordinarily need to use this method.

See also: – **setLanguage**

setAccessoryView:

– **setAccessoryView:***aView*

Adds *aView* to the contents of the Spelling panel. An application can invoke this method to add controls that extend the panel's functions. The Spelling panel is automatically resized to accommodate *aView*. This method can be invoked repeatedly to change the accessory View depending on the situation. When *aView* is **nil**, the effect is to remove any accessory View that's already in the panel. Returns the former accessory view, or **nil** if there was none.

See also: – **accessoryView**

setIgnoredWords:forSpellDocument:

– **setIgnoredWords:**(const char *const *)*someWords* **forSpellDocument:**(int)*tag*

Initializes the NXSpellChecker's list of acceptable words for the document identified by *tag*. The first argument is an array of pointers to the words. The second argument identifies the document for which the list is being maintained (described above in the section "Matching a List of Ignored Words with the Document It Belongs To").

See also: – **spellDocumentTag** (NXIgnoreMisspelledWords protocol)

setLanguage:

– **setLanguage:**(const char *)*aLanguage*

Tells the NXSpellChecker object what language to use in subsequent spell check requests. This method is needed only if the application sometimes overrides the language established by the user's system defaults or by the user's choice of dictionary in the Spelling panel. (That might be required while checking a document whose text is distributed among several objects, each in a different language). Upon completion of the check in a different language, the application should restore the default. To do so, before using **setLanguage:**, use **language** to get the language previously in effect, and afterwards use **setLanguage:** to restore it.

Setting a different language causes corresponding changes to the selected dictionary and to the user's list of acceptable words. Suppose that "checker" is the **id** of an NXSpellChecker instance, and the application sends the following messages:

```
[checker setLanguage:"French"]
[checker checkSpelling:how of:textToBeChecked]
```

During the check invoked by the second message, the spelling system refers to the French dictionary rather than the dictionary selected in the Spelling panel, and adds "learned" words to the user's French word list.

If *aLanguage* is NULL, sets the language to the first language for which there is a dictionary from the list of languages returned by **systemLanguages** (which reports defaults the user has set for the active application, and if none have been set, the user's global language preference). Returns **self** when a language has been set, **nil** otherwise.

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

spellingPanel

– **spellingPanel**

Returns the Panel in which the user can choose a dictionary or select actions with respect to misspelled words. You will need to identify the Panel in order to bring it forward when the user selects **Spelling...** in the application's **Services** menu. You may also need it if your application elects to modify the panel with **setAccessoryView:**.

See also: – **setAccessoryView:**

NXSpellServer

Inherits From: Object

Declared In: appkit/NXSpellServer.h

The NXSpellServer class gives you a way to make your particular spelling checker a service that's available to any application. A *service* is an application that declares its availability in a standard way, so that any other applications that wish to use it can do so. (See the discussion of "Services" under "Other Features," later in this chapter.)

The spelling checker bundled with NeXTSTEP makes itself available in this way. If you build a spelling checker that makes use of the NXSpellServer class and list it as an available service, then users of any application that makes use of NXSpellChecker or includes a Services menu will see your spelling checker as one of the available dictionaries, along with the one provided by NeXT.

To make use of NXSpellServer, you write a small program that creates an NXSpellServer instance and a delegate that responds to messages asking it to find a misspelled word and to suggest guesses for a misspelled word. Send the NXSpellServer **registerLanguage:** messages to tell it the languages your delegate can handle.

The program that runs your spelling checker should not be built as an Application Kit application, but as a simple program. Suppose you supply spelling checkers under the vendor name "Acme." Suppose the file containing the code for your delegate is called AcmeEnglishSpellChecker. Then the following might be your program's **main**:

```
void main()
{
  NXSpellServer *aServer = [[NXSpellServer alloc] init];
  if ([aServer registerLanguage:"English" byVendor:"Acme"]) {
    [aServer setDelegate:[AcmeEnglishSpellChecker new]];
    [aServer run];
    fprintf(stderr, "Unexpected death of Acme SpellChecker!\n");
  } else {
    fprintf(stderr, "Unable to check in Acme SpellChecker.\n");
  }
}
```

Your delegate is an instance of a custom subclass. (It's simplest to make it a subclass of Object, but that's not a requirement.) Given a text stream, your delegate must be able to find a misspelled word by implementing the method **spellServer:findMisspelledWord:length:inLanguage:inTextStream:startingAt:wordCount:countOnly:**. Usually, this method also reports the number of words it has scanned, but that isn't mandatory.

Optionally, the delegate may also suggest corrections for misspelled words. It does so by implementing the method **spellServer:suggestGuessesForWord:inLanguage:**

Service Availability Notice

When there's more than one spelling checker available, the user selects the one desired. The application that requests a spelling check uses an NXSpellChecker object, and it provides a Spelling Panel; in the panel there's a pop-up list of available spelling checkers. Your spelling checker appears in that list if it has a *service descriptor.*

A service descriptor is an entry in a text file called **services**. Usually it's located within the bundle that also contains your spelling checker's executable file. The bundle (or directory) that contains the services file must have a name ending in ".service" or ".app". The system looks for service bundles in the directories **~/Apps**, **/LocalApps**, and **/NextApps**.

A service availability notice has a standard format, illustrated in the following example for the Acme spelling checker:

```
Spell Checker:  Acme
Language:  French
Language:  English
Executable:  franglais.daemon
```

The first line identifies the type of service; for a spelling checker, it must say "Spell Checker:" followed by your vendor name. The next line contains the English name of a language your spelling checker is prepared to check. The language should be one of those registered with NeXT Developer Support. If your program can check more than one language, use an additional line for each additional language. The last line of a descriptor gives the name of the service's executable file. (It requires a complete path if it's in a different directory.)

When there's a service descriptor for your Acme spelling checker and also a service descriptor for the checker provided with NeXTSTEP, a user looking at the Spelling Panel's pop-up list would see:

```
English (Acme)
English (NeXT)
French (Acme)
```

Illustrative Sequence of Messages to an NXSpellServer

The act of checking spelling usually involves the interplay of objects in two classes: the user application's NXSpellChecker (which responds to interactions with the user) and your spelling checker's NXSpellServer (which provides the application interface for your spelling checker). You can see the interaction between the two in the following list of steps involved in finding a misspelled word.

- The user of an application selects a menu item to request a spelling check. The application sends a message to its NXSpellChecker object. The NXSpellChecker in turn sends a corresponding message to the appropriate NXSpellServer.
- The NXSpellServer receives the message asking it to check the spelling of a text stream. It forwards the message to its delegate.
- The delegate searches for a misspelled word. If it finds one, it returns YES and identifies the word's location in the text stream.
- The NXSpellServer receives a message asking it to suggest guesses for the correct spelling of a misspelled word, and forwards the message to its delegate.
- As the delegate finds each possible correction, it sends an **addGuess:** message to the NXSpellServer, causing it to append each new word to a list of guesses. When the delegate method returns, the NXSpellServer returns the completed list to the NXSpellChecker that initiated the request.
- The NXSpellServer doesn't know what the user does with the errors its delegate has found or with the guesses its delegate has proposed. (Perhaps the user corrects the document, perhaps by selecting a correction from the NXSpellChecker's display of guesses; but that's outside the NXSpellServer's purview.) However, if the user presses the Learn or Forget buttons (thereby causing the NXSpellChecker to revise the user's word list), the NXSpellServer receives a notification of the word thus learned or forgotten. It's up to you whether your spell checker acts on this information. If the user presses the Ignore button, the NXSpellServer is not notified (but the next time that word occurs in the text, the method **isInUserDictionary:caseSensitive:** will report YES rather than NO).
- Once the NXSpellServer delegate has reported a misspelled word, it has completed its search. Of course, it's likely that the user's application will then send a new message, this time asking the NXSpellServer to check a text stream that is in fact the part of the text it didn't get to earlier.

Method Types

Setting the delegate	– setDelegate: – delegate
Registering your service	– registerLanguage:byVendor:
Starting your service	– run
Checking user dictionaries	– isInUserDictionary:caseSensitive:
Receiving alternatives	– addGuess:

Instance Methods

addGuess:

– **addGuess:**(const char *)*guess*

Appends a word to the list of possible corrections for a misspelled word. The delegate's implementation of **spellServer:suggestGuessesForWord:inLanguage:** should invoke this method in order to append each new guess that it finds to the list that the NXSpellServer is compiling.

delegate

– **delegate**

Returns the NXSpellServer's delegate.

See also: – **setDelegate:**

isInUserDictionary:caseSensitive:

– (BOOL)**isInUserDictionary:**(const char *)*word* **caseSensitive:**(BOOL)*flag*

Reports whether a word is in the user's list of learned words, or the document's list of words to ignore. The first argument is a word to be checked. The second is YES when the comparison is to be case-sensitive.

Returns YES if the word is acceptable to the user.

registerLanguage:byVendor:

– (BOOL)**registerLanguage:**(const char *)*language* **byVendor:**(const char *)*vendor*

Notifies the NXSpellServer of a language your spelling checker can check. The argument *language* is the English name of a language on NeXT's list of languages. The argument *vendor* identifies the vendor (to distinguish your spelling checker from those that others may offer for the same language). If your spelling checker supports more than one language, it should invoke this method once for each language. Registering a language/vendor combination causes it to appear in the Spelling Panel's pop-up labeled "Dictionary". Returns YES when the language is registered, NO if for some reason it can't be registered.

run

– **run**

Starts a loop in which the NXSpellServer awaits requests for its services. This loop normally runs forever.

setDelegate:

– **setDelegate:***anObject*

Appoints the object identified in the argument as the delegate of your NXSpellServer. Since the delegate is where the real work is done, this is an essential step before your program sends the NXSpellServer its **run** message. Returns the delegate.

See also: – **delegate**

Methods Implemented by the Delegate

The real work of checking is done not by the NXSpellServer but by its delegate. The method **sender:findMisspelledWord:...** does the actual checking. The method **sender: suggestGuessesForWord:...** is optional; if implemented, it supplies a list of possible corrections for a misspelled word.

spellServer:didForgetWord:inLanguage:

– (void)**spellServer:**(NXSpellServer *)*sender*
didForgetWord:(const char *)*word*
inLanguage:(const char *)*language*

Notification to the NXSpellServer's delegate that the user has pressed **Forget** in an NXSpellChecker's Spelling Panel (and presumably the NXSpellChecker has removed the word from the user's list of acceptable words). If the delegate maintains a similar auxiliary word list, it may wish to edit its list accordingly.

See also: – **spellServer:didLearnWord:inLanguage:**

spellServer:didLearnWord:inLanguage:

– (void)**spellServer:**(NXSpellServer *)*sender*
didLearnWord:(const char *)*word*
inLanguage:(const char *)*language*

Notification to the NXSpellServer's delegate that the user has pressed **Learn** in an NXSpellChecker's Spelling Panel (and presumably the NXSpellChecker has removed the word from the user's list of acceptable words). If the delegate maintains a similar auxiliary word list, it may wish to edit it accordingly.

See also: – **spellServer:didForgetWord:inLanguage:**

spellServer:findMisspelledWord:length:inLanguage:inTextStream: startingAt:wordCount:countOnly:

– (BOOL)**spellServer:**(NXSpellServer *)*sender*
findMisspelledWord:(int *)*start*
length:(int *)*length*
inLanguage:(const char *)*language*
inTextStream:(id <NXReadOnlyTextStream>)*textStream*
startingAt:(int)*startPosition*
wordCount:(int *)*number*
countOnly:(BOOL)*flag*

Searches the text stream for a misspelled word. *textStream* identifies the text stream to be checked. *startPosition* is the offset of the current character. *language* identifies the language of the text stream.

start, *length*, and *number* are pointers to values that the method will set. The method identifies a misspelled word by putting its offset in the text stream into *start* and its length into *length*. These values (like *startPosition*) are the number of characters, which may be less than the offset in bytes if the text stream contains multibyte characters. The method puts the number of words it has checked into *number*. Thus, *number* will contain the number of words that precede the misspelled word, or the number of words in the entire text end if no word is misspelled. If for some reason the method is unable to count words, it should set *number* to –1.

When *flag* is YES, the method simply counts the words from *startingPoint* to the end of the text stream, without checking their spelling.

When the method finds a misspelled word, it should then invoke the NXSpellServer's method **isInUserDictionary:caseSensitive:** to discover whether the word is acceptable to the user or to the document. It should end its search and return YES only if it has found a word that is not acceptable to either of them.

Returns YES if a misspelled word has been found, and sets *start* and *length* to identify the misspelling. Returns NO if the search reaches the end of the text stream without encountering a misspelled word, or whenever *flag* is YES.

spellServer:suggestGuessesForWord:inLanguage:

– (void)**spellServer:**(NXSpellServer *)sender
suggestGuessesForWord:(const char *)*word*
inLanguage:(const char *)*language*

Searches for words that (by whatever criterion it chooses to adopt) seem possible corrections for the misspelled example it receives as *word*. For each candidate that it finds, it sends an **addGuess:** message to the NXSpellServer object, which takes care of accumulating the suggested words.

NXSplitView

Inherits From: View : Responder : Object

Declared In: appkit/NXSplitView.h

Class Description

An NXSplitView object lets several Views share a region within a window. The NXSplitView resizes its subviews so that each subview is the same width as the NXSplitView, and the total of the subviews' heights is equal to the height of the NXSplitView. The NXSplitView positions its subviews so that the first subview is at the top of the NXSplitView, and each successive subview is positioned below. The user can set the height of two subviews by moving a horizontal bar called the *divider*, which makes one subview smaller and the other larger.

To add a View to an NXSplitView, you use the **addSubview:** View method. When the NXSplitView is displayed, it checks to see if its subviews are properly tiled. If not, it invokes the **splitView:resizeSubviews:** delegate method, allowing the delegate to specify the heights of specific subviews. If the delegate doesn't implement a **splitView:resizeSubviews:** method, **the NXSplitView sends adjustSubviews** to itself to yield the default tiling behavior.

When a mouse-down occurs in an NXSplitView's divider, the NXSplitView determines the limits of the divider's travel and tracks the mouse to allow the user to drag the divider within these limits. With the following mouse-up, the NXSplitView resizes the two affected subviews, informs the delegate that the subviews were resized, and displays the affected Views and divider. The NXSplitView's delegate can constrain the travel of specific dividers by implementing the **splitView:getMinY:maxY:ofSubviewAt:** method.

Instance Variables

id **delegate**;

delegate	The object that receives notification messages from the NXSplitView.

Method Types

Initializing an NXSplitView	– initFrame:
Handling Events	– mouseDown: – acceptsFirstMouse
Managing component Views	– adjustSubviews – resizeSubviews: – dividerHeight – drawSelf:: – drawDivider: – setAutoresizeSubviews:
Assigning a delegate	– delegate – setDelegate:

Instance Methods

acceptsFirstMouse

– (BOOL)**acceptsFirstMouse**

Returns YES, thus allowing the NXSplitView to respond to the mouse event that made its Window the key window.

See also: – **acceptsFirstMouse** (View)

adjustSubviews

– **adjustSubviews**

Adjusts the heights of the NXSplitView's subviews so the total height fills the NXSplitView. The subviews are resized proportionally; the size of a subview relative to the other subviews doesn't change. This method is invoked if the NXSplitView's delegate doesn't respond to a **splitView:resizeSubviews:** message. Returns **self**.

See also: – **setDelegate:**, – **splitView:resizeSubviews:** (delegate method), – **setFrame:** (View)

delegate

– **delegate**

Returns the NXSplitView's delegate.

See also: – **setDelegate:**

dividerHeight

– (NXCoord)**dividerHeight**

Returns the height of the divider. You can override this method to change the divider's height, if necessary.

See also: – **drawDivider:**

drawDivider:

– **drawDivider:**(const NXRect *)*aRect*

Draws a divider between two of the NXSplitView's subviews. *aRect* describes the entire divider rectangle in the NXSplitView's coordinates, which are flipped. The default implementation composites a default image to the center of *aRect*; if you override this method and use a different icon to identify the divider, you may want to change the height of the divider. Returns **self**.

See also: – **dividerHeight** – **composite:toPoint:** (NXImage)

drawSelf::

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

Draws the NXSplitView. You never invoke this method directly; it's invoked by the display mechanism. Returns **self**.

See also: – **drawDivider:**, – **resizeSubviews:**, – **display:** (View)

initFrame:

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

Initializes the NXSplitView, which must be a newly allocated NXSplitView instance, setting its frame rectangle to the argument. The NXSplitView's coordinate system is flipped, and it's set to autoresize its subviews. This method is the designated initializer for the NXSplitView class. Returns **self**.

See also: – **setAutoresizeSubviews:** (View)

mouseDown:

– **mouseDown:**(NXEvent *)*theEvent*

You never invoke this method; it's invoked when the user clicks in the NXSplitView. Returns **self**.

See also: – **splitView:getMinY:maxY:ofSubviewAt:** (delegate), – **splitViewDidResizeSubviews:** (delegate), – **setFrame:** (View)

resizeSubviews:

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

Ensures that the NXSplitView's subviews are properly sized to fill the NXSplitView. If the delegate implements the **splitView:resizeSubviews:** method, that method is invoked to resize the subviews; otherwise, the **adjustSubviews** method is invoked to resize the subviews. In either case, this method then informs the delegate that the subviews were resized. *oldSize* is the previous bounds rectangle size. Returns **self**.

See also: – **splitView:resizeSubviews:** (delegate), – **adjustSubviews**, – **splitViewDidResizeSubviews:** (delegate), – **resizeSubviews:** (View)

setAutoresizeSubviews:

– **setAutoresizeSubviews:**(BOOL)*flag*

Overrides View's **setAutoresizeSubviews:** method to ensure that automatic resizing of subviews will not be disabled. You should never invoke this method. Returns **self**.

setDelegate:

– **setDelegate:***anObject*

Makes *anObject* the NXSplitView's delegate. The notification messages that the delegate can expect to receive are listed at the end of the NXSplitView class specifications. The delegate doesn't need to implement all the delegate methods. Returns **self**.

See also: – **delegate**

Methods Implemented by the Delegate

splitView:getMinY:maxY:ofSubviewAt:

– **splitView:***sender*
getMinY:(NXCoord *)*minY*
maxY:(NXCoord *)*maxY*
ofSubviewAt:(int)*offset*

Allows the delegate to constrain the *y* coordinate limits of a divider when the user drags the mouse. This method is invoked before the NXSplitView begins tracking the mouse to position a divider. When this method is invoked, the limits have already been set and are stored in *minY* (the topmost limit) and *maxY* (the bottommost limit). You may further constrain the limits by setting the variables indicated by *minY* and *maxY*, but you cannot extend the divider limits. *minY* and *maxY* are specified in the NXSplitView's flipped coordinate system. The divider to be repositioned is indicated by *offset*, an index that counts the dividers from top to bottom starting with divider 0.

See also: – **mouseDown:**

splitView:resizeSubviews:

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

Allows the delegate to specify custom sizing behavior for the subviews of the NXSplitView. If the delegate implements this method, **splitView:resizeSubviews:** is invoked after the NXSplitView is resized; otherwise, **adjustSubviews** is invoked to retile the subviews. The old size of the NXSplitView is indicated by *oldSize*; the subviews should be resized such that the sum of the heights of the subviews plus the sum of the heights of the dividers equals the height of the NXSplitView's new frame. You can get the height of a divider through the **dividerHeight** method.

See also: – **adjustSubviews**, – **dividerHeight**, – **setFrame:** (View)

splitViewDidResizeSubviews:

– **splitViewDidResizeSubviews:***sender*

Informs the delegate that the sizes of some or all of the NXSplitView's subviews were changed. This method is invoked when the NXSplitView resizes all its subviews because its frame rectangle changed, and also after the NXSplitView resizes two subviews in response to the repositioning of a divider.

See also: – **resizeSubviews:**, – **mouseDown:**

Object Additions

Inherits From: none (*Object is the root class.*)

Declared In: appkit/Application.h

Class Description

The Application Kit adds one method, **perform:with:afterDelay:cancelPrevious:**, to the root Object class. This method becomes part of the class for all applications that use the Kit, but not for applications that don't.

Only this one method is described here. See Chapter 1, "Root Class," for a general description of the Object class and the methods it defines.

Instance Methods

perform:with:afterDelay:cancelPrevious:

– **perform:**(SEL)*aSelector*
with:*anObject*
afterDelay:(int)*ms*
cancelPrevious:(BOOL)*flag*

Registers a timed entry to send an *aSelector* message to the receiver after a delay of at least *ms* milliseconds, provided *ms* is 1 or greater. This method returns before the *aSelector* message is sent. However, if *ms* is 0, a timed entry is not registered and the message is sent immediately, before this method returns. In either case, it returns **self**.

The timed entry that this method registers can be called only after the application finishes responding to the current event and is ready to get the next event. Therefore, program activity could delay the message well beyond *ms* milliseconds. The timed entry is registered at a priority of NX_RUNMODALTHRESHOLD, which means that it can be called when getting an event in the main event loop or in a modal event loop for an attention panel, but not during a modal loop for a button, slider, or other control device.

The *aSelector* method should not have a significant return value and should take a single argument of type **id**; *anObject* will be the argument passed in the message.

If *flag* is YES and another **perform:with:afterDelay:cancelPrevious:** message is sent to the same receiver to have it perform the same *aSelector* method, the first request to perform the *aSelector* method is canceled. Thus successive **perform:with:afterDelay:cancelPrevious:** messages can repeatedly postpone the *aSelector* message.

If *flag* is NO, each **perform:with:afterDelay:cancelPrevious:** message will cause another delayed *aSelector* message to be sent.

This method permits you to register an action in response to a user event (such as a click), but delay it in case subsequent events alter the environment in which the action would be performed (for example, if the click turns out to be double-click). It can also be used to postpone a message that updates a display until after a number of changes have accumulated, or to delay a **free** message to an object until after the application has finished responding to the current event. (Application's **delayedFree:** method offers another way to delay **free** messages.)

See also: – **perform:with:** (Object), – **delayedFree:** (Application class)

OpenPanel

Inherits From: SavePanel : Panel : Window : Responder : Object

Declared In: appkit/OpenPanel.h

Class Description

The OpenPanel provides a convenient way for an application to query the user for the name of a file to open. It can only be run modally. (The user should use the directory browser in the Workspace for non-modal opens.) It allows you to specify the types of candidate files whose names will appear in the OpenPanel, and then to filter-out unwanted file types.

Every application has one and only one OpenPanel, and the **new** method returns a pointer to it. Do not attempt to create a new OpenPanel using the methods **alloc** or **allocFromZone**; these methods are inherited from SavePanel, which overrides them to return errors if used.

See the class description for SavePanel for more information.

Instance Variables

char ****filterTypes**;

filterTypes	File types allowed to open

Method Types

Creating and Freeing an OpenPanel	+ new + newContent:style:backing:buttonMask:defer: – free
Setting the OpenPanel class	+ setOpenPanelFactory:
Filtering files	– allowMultipleFiles:

Querying the chosen files	– filenames
Choosing directories	– chooseDirectories:
Running the OpenPanel	– runModalForDirectory:file: – runModalForDirectory:file:types: – runModalForTypes:

Class Methods

new

+ **new**

Creates, if necessary, and returns the shared instance of OpenPanel. Each application has just one instance of OpenPanel. This method is implemented to override the inherited **new** method to assure that only one instance of OpenPanel is created in an application.

newContent:style:backing:buttonMask:defer:

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

Don't use this method, invoke **new** instead. This method is implemented to override the **newContent:style:backing:buttonMask:defer:** method inherited from SavePanel. Returns **self**.

See also: + new

setOpenPanelFactory:

+ **setOpenPanelFactory:***class*

Sets the class from which OpenPanel will be instantiated. *class* should be a subclass of OpenPanel. An application must invoke this method before it creates the shared instance of OpenPanel. When the **new** method is invoked, the object it returns will belong to *class*.

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

Instance Methods

allowMultipleFiles:

– **allowMultipleFiles:**(BOOL)*flag*

If *flag* is YES, then the user can select more than one file in the browser. If multiple files are allowed, then the **filename** method—inherited from SavePanel—returns a non-NULL value only if one and only one file is selected. By contrast, OpenPanel's **filenames** method always returns the selected files, even if only one file is selected. A further distinction between the two methods is that the inherited **filename** method always returns a fully-specified path, while the **filenames** method doesn't; the names it returns are always relative to the path returned by **directory**. Returns **self**.

See also: – **directory**, – **filename** (SavePanel), – **filenames**

chooseDirectories:

– **chooseDirectories:**(BOOL)*flag*

Sets the OpenPanel to get directory names from the user. Invoke this method before running the panel. If *flag* is YES, the OpenPanel's filterTypes are ignored and only directories will appear in the OpenPanel file browser. If *flag* is NO (the default), the OpenPanel allows the user to select files or directories, and filterTypes are used.

See also: – runModalForDirectory:file:types:

filenames

– (const char *const *)**filenames**

Returns a NULL terminated list of files (relative to the path returned by **directory**). This list will be valid even if **allowMultipleFiles** is NO, in which case this method returns a single entry. This is the preferred method to get the name or names of any files that the user has chosen.

See also: – **directory**, – **filename** (SavePanel)

free

– **free**

Frees the storage used by the shared OpenPanel object and returns **nil**. The next time **new** is sent to the OpenPanel, it will be recreated. You probably never need to invoke this method since there is one shared instance of the OpenPanel.

See also: + **new**

runModalForDirectory:file:

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

Initializes the panel to the file specified by *path* and *filename*, then displays it and begins its modal event loop. Invokes the superclass's corresponding method, which invokes Application's **runModalFor:** method with **self** as the argument. Returns the constant returned by the **runModalFor:** method, depending on the method used to stop the modal event loop.

See also: – **runModalFor:** (Application), – **runModalForDirectory:file:** (SavePanel)

runModalForDirectory:file:types:

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

Loads the directory specified in *path* and optionally sets *filename* as the default file to open. If *filename* is NULL, no default file is set. *fileTypes* is a NULL-terminated list of extensions (not including the period) to be used to filter candidate files. If the first item in the list is a NULL, then all ASCII files will be included.

Invokes the **runModalForDirectory:file:** method and returns the value returned by that method.

See also: – **runModalForDirectory:file:**

runModalForTypes:

– (int)**runModalForTypes:**(const char *const *)*fileTypes*

Invokes the **runModalForDirectory:file:types:** method, using the last directory from which a file was chosen as the *path* argument. Returns the value returned by that method.

See also: – **runModalForDirectory:file:types:**

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. The PageLayout panel is created, displayed, and run (in a modal loop) when a **runPageLayout:** message is sent to the Application object. By default, this message is sent up the responder chain when the user clicks the Page Layout menu item.

Each application can have but one PageLayout object. If you're creating a subclass of PrintPanel, you should send your subclass' class object a **new** message (without invoking **alloc** or **allocFromZone:**) before any **runPageLayout:** messages are sent to ensure that an instance of your subclass is the unique PrintPanel object for your application.

You can add your own controls to the Page Layout panel through the **setAccessoryView:** method. The panel is automatically resized to accommodate the View that you've added. Note that you can't retrieve the PageLayout's settings through messages to the object. If the controls that you add depend on the values of the existing controls (or vice versa), you must subclass PageLayout and query or set these controls by sending messages to the instance variables that represent them.

Instance Variables

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 portrait/landscape Matrix object.
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.
accessoryView	The optional View added by the application.

Method Types

Creating and freeing a PageLayout instance	+ new + newContent:style:backing:buttonMask:defer: – free
Running the PageLayout panel	– runModal
Customizing the 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.

allocFromZone:

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

new

+ **new**

Returns the application's sole PageLayout object, creating it if necessary. This method is invoked by Application's **runPageLayout:** method; by extension, it's invoked when the user clicks the Page Layout menu item.

newContent:style:backing:buttonMask:defer:

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

Initializes the PageLayout object. You never invoke this method directly; use **new** instead.

Instance Methods

accessoryView

– **accessoryView**

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

See also: – **setAccessoryView:**

convertOldFactor:newFactor:

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

The standard unit used to measure a paper's dimensions is a point (for example, the PrintInfo object defines a paper's size in units of points). This method returns, by reference, a value that expresses the ratio between a point and the currently chosen unit of measurement. In general, both *old* and *new* are set to this value. The only time the values returned in the arguments differ is when the unit of measurement is being changed. Specifically, if you invoke this method from within **pickedUnits:**, *old* gives the old ratio and *new* gives the new one. Returns **self**.

See also: – **pickedUnits:**

free

– **free**

Frees the PageLayout object and its contents, including the accessory View.

pickedButton:

– **pickedButton:***sender*

The action of the OK and Cancel buttons, this method ends the Page Layout panel's modal run. If the OK button inspired this method, the height, width, and scale entries must be acceptable (they must hold positive numbers), otherwise the unacceptable entry is selected and the panel isn't stopped. If the panel is being cancelled, then it's stopped regardless of the entries' acceptability. If the panel is successfully stopped, the **exitTag** instance variable is set to the sender's tag (NX_OKTAG or NX_CANCELTAG). Returns **self**.

pickedLayout:

– **pickedLayout:***sender*

Performed when the user selects an item from the Layout list. You can get the new layout with the message

```
[[sender selectedCell] title]
```

Returns **self**.

pickedOrientation:

– **pickedOrientation:***sender*

Performed when the user selects a page orientation from the Portrait/Landscape matrix. This method updates the Width and Height fields, and redraws the paper view. You can get the new orientation by sending the message

```
int orientation = [sender selectedCol]
```

and comparing the returned value to NX_LANDSCAPE and NX_PORTRAIT. Returns **self**.

pickedPaperSize:

– pickedPaperSize:*sender*

Performed when the user selects a paper size from the Paper Size list. This method updates the Width and Height fields, redraws the paper view, and may switch the Portrait/Landscape orientation. The following demonstrates how to retrieve the name of a paper size and an NXSize describing the paper's dimensions from the **paperSizeList** instance variable:

```
const char *paperName = [[paperSizeList selectedCell] title];
const NXSize *paperSize = NXFindPaperSize(paperName);
```

Returns **self**.

pickedUnits:

– pickedUnits:*sender*

Performed when the user selects a new unit of measurement from the Units list. The height and width fields are updated. Controls in the accessory view that express dimensions on the page must be converted to the new unit of measurement. The ratios returned by **convertOldFactor:newFactor:** method should be used to calculate the new values, as shown below. In the example, a hypothetical PageLayout subclass uses a TextField (**myField**) to display a value measured in the chosen units:

```
- 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 myField based on the conversion factors. */
    [myField setFloatValue:([myField floatValue] * new / old)];

    /* Set the selected units. */
    return [super pickedUnits:sender];
}
```

Returns **self**.

See also: **– convertOldFactor:newFactor:**

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 shouldn't need to invoke it yourself. Returns **self**.

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

runModal

– (int)**runModal**

Reads the pertinent data from the PrintInfo object into the PageLayout object and then runs the Page Layout panel in a modal loop. When the user clicks the Cancel or OK button the loop is broken (from within the **pickedButton:** method), the panel is hidden, and, if the button was OK, the new PageLayout values are written to the PrintInfo object. The method returns the tag of the button that the user clicked to dismiss the panel (either NX_OKTAG or NX_CANCELTAG).

This method is invoked by Application's **runPageLayout** method; an application is best served by running the Page Layout panel from that method rather than invoking this one directly.

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

setAccessoryView:

– **setAccessoryView:***aView*

Adds *aView* to the PageLayout's view hierarchy. Applications can invoke this method to add a View that contains their own controls. The 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**, the panel's current accessory view, if any, is removed. Returns the old accessory view.

See also: – **accessoryView**

textDidEnd:endChar:

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

Performed when the user finishes typing in the Height or Width forms. The Paper Size list and Orientation matrix may change. You can override this method to update other controls you add to the panel. Returns **self**.

textWillChange:

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

You never invoke this method directly; it's invoked 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.

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**

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, TextFields, Sliders, and so on. Menu is a sublcass of Panel.

Panels behave differently from 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

None declared in this class.

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 return value of NO continues the **commandKey:** message down the Application object's list of Windows; a return value of YES 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 a View that can become the first responder. The default return value 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 buffered. 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. The Panel has no title.

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 receives **keyDown:** event messages only when it's the key window and either:

- none of its Views is the first responder,
- none of the Views in its responder chain implements a **keyDown:** method, or
- the Views in its responder chain that implement a **keyDown:** method include the message [super keyDown:theEvent].

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 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 conditions should be true 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)**worksWhenModal**

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:**

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 five standard pasteboards in common use:

General pasteboard	The pasteboard that's used for ordinary cut, copy, and paste operations. It holds the contents of the last selection that's been cut or copied.
Font pasteboard	The pasteboard that holds font and character information and supports 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.
Drag pasteboard	The pasteboard that stores data to be manipulated as the result of a drag operation.

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

NXGeneralPboard
NXFontPboard
NXRulerPboard
NXFindPboard
NXDragPboard

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 never more than 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.

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 (of type NXAtom) are string pointers for the standard NeXT pasteboard types. This list is not exhaustive; NeXTSTEP kits define some other data types that can only be read by kit objects and are not intended for general purpose data interchange.

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
NXFileContentsPboardType	A representation of a file's contents
NXColorPboardType	NXColor data
NXSelectionPboardType	Describes a selection
NXDataLinkPboardType	Defines a link between documents

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

Reading and Writing Data

Typically, data is written to the pasteboard using **writeType:data:length:** and read using **readType:data:length:**. However, file contents and colors must be written using special pasteboard methods:

- Data of NXFileContentsPboardType, representing the contents of a named file, must be written using **writeFileContents:** and read using **readFileContentsType:toFile:**.
- NXColor data should be written using the **NXWriteColorToPasteboard**() function, and read using **NXReadColorFromPasteboard**().

It's often convenient (and most memory-efficient) to prepare data for the pasteboard by writing data to a memory stream through functions such as **NXWrite**(), **NXPrintf**(), and **NXPutc**(). After the data has been written, the stream can be sent to the pasteboard server using **writeType:fromStream:**.

Similarly, you can get a memory stream for the data received from the pasteboard server via **readTypeToStream:** 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 raises an NX_pasteboardComm exception.

Instance Variables

id **owner**;

owner	The object responsible for putting data in the pasteboard.

Method Types

Creating and freeing a Pasteboard object	+ new + newName: + newUnique – free – freeGlobally
Getting data in different formats	+ newByFilteringFile: + newByFilteringData:ofType: + newByFilteringTypesInPasteboard: + typesFilterableTo:
Referring to a Pasteboard by name	+ newName: – name
Writing data	– declareTypes:num:owner: – addTypes:num:owner: – writeType:data:length: – writeType:fromStream: – writeFileContents:
Discerning types	– types – findAvailableTypeFrom:num:

Reading data	– changeCount – readType:data:length: – readTypeToStream: – readFileContentsType:toFile: – deallocatePasteboardData: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 NXGeneralPboard to the **newName:** method.

newByFilteringData:ofType:

+ **newByFilteringData:**(NXData *)*data* **ofType:**(const char *)*type*

Creates and returns a new Pasteboard with a unique name that has, declared within it, data of every type that can be provided by the available filter services from *data*. The returned pasteboard also declares data of the supplied type *type*. No filter service is invoked until the data is actually requested, so invoking this method is reasonably inexpensive.

newByFilteringFile:

+ **newByFilteringFile:**(const char *)*filename*

Creates and returns a new Pasteboard with a unique name that has, declared within it, data of every type that can be provided by the available filter services from the file *filename*. No filter service is invoked until the data is actually requested, so invoking this method is reasonably inexpensive.

newByFilteringTypesInPasteboard:

+ **newByFilteringTypesInPasteboard:**(Pasteboard *)*pboard*

Creates and returns a new Pasteboard with a unique name that has, declared within it, data of every type that can be provided by the available filter services from the data on pasteboard *pboard*. This process can be thought of as expanding the pasteboard, since the new pasteboard generally will contain more representations of the data on *pboard*.

This method returns *pboard* if *pboard* is a pasteboard returned by one of the **newByFiltering...** methods, so a pasteboard can't be expanded multiple times. This method only returns the original types and the types that can be created as a result of a single filter; the pasteboard will not have defined types that are the result of translation by multiple filters.

No filter service is invoked until the data is actually requested, so invoking this method is reasonably inexpensive.

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:

NXGeneralPboard
NXFontPboard
NXRulerPboard
NXFindPboard
NXDragPboard

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

newUnique

+ **newUnique**

Creates and returns a new Pasteboard with a name that is guaranteed to be unique with respect to other Pasteboards on the system. This method is useful for applications that implement their own interprocess communication using pasteboards.

typesFilterableTo:

+ (NXAtom *)**typesFilterableTo:**(const char *)*type*

Returns a null-terminated array of NXAtoms indicating the types that data of type *type* can be converted to by available filter services. The array contains the original type. The caller is responsible for freeing the returned array.

Instance Methods

addTypes:num:owner:

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

Adds additional types to the pasteboard. This method can be useful when multiple entities (such as a combination of application and library methods) contribute data for a single copy command. It should only be invoked after a **declareTypes:num:owner:** message has been sent for the same data. The owner for the new types may be different from the owner(s) of the previously declared data.

Returns the pasteboard's change count, or 0 in case of an error.

See also: – **changeCount**

changeCount

– (int)**changeCount**

Returns the current change count for the pasteboard. The change count is a system-wide variable that increments every time the contents of the pasteboard changes (a new owner is declared). By examining the change count, an application can determine whether the current data in the pasteboard is the same as the data it last received.

An independent change count is maintained for each named pasteboard.

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

deallocatePasteboardData:length:

– **deallocatePasteboardData:**(char *)*data* **length:**(int)*numBytes*

This method should be used to deallocate the memory returned by **readType:data:length:**. Returns **self** if the memory is successfully deallocated, otherwise raises an NX_appkitVMError exception.

declareTypes:num:owner:

– (int)**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 (typically, the richest representation is first).

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 the pasteboard's change count.

See also: – **writeType:data:length:**, – **pasteboard:provideData:**, – **addTypes:num:owner:**, – **changeCount**

findAvailableTypeFrom:num:

– (const char *)**findAvailableTypeFrom:**(const char *const *)*types*
num:(int)*numTypes*

Scans the types defined by *types* (which is an array of size *numTypes*) and returns the first type that matches a type declared on the pasteboard. A **types** or **findAvailableTypeFrom:num:** message should be sent before reading any data from the pasteboard.

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

– (NXAtom)**name**

Returns the name of the Pasteboard object.

See also: + newName:

readFileContentsType:toFile:

– (char *)**readFileContentsType:**(const char *)*type* **toFile:**(const char *)*filename*

Reads data representing a file's contents from the pasteboard, and writes it to the file *filename*. Data of any file contents type should only be read using this method. *type* should generally be specified; if *type* is NULL, a type based on *filename*'s extention (as returned

by **NXCreateFileContentsPboardType()**) is substituted. If data matching *type* isn't found on the pasteboard, data of type NXFileContentsPboardType is requested. Returns an allocated string with the name of the file that the data was actually written to.

See also: – writeFileContents:

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 shouldn't attempt to use the pointer referred to by *theData*, as it won't be valid.

The memory for the data that this method provides must eventually be freed by the caller using **deallocatePasteboardData:length:**; you should *not* attempt to free the returned memory using **vm_deallocate()** or **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 */
    [myPasteboard deallocatePasteboardData:data length:length];
}
```

See also: – readTypeToStream:

readTypeToStream:

– (NXStream *)**readTypeToStream:**(const char *)*dataType*

Reads data from the pasteboard to a stream. This method uses the **readType:data:length:** method to read data of the type *dataType* from the pasteboard. It then opens a stream on the data, and returns the stream, or NULL if there is an error. Data returned with this method must eventually be freed using

```
NXCloseMemory(theStream, NX_FREEBUFFER)
```

You should *not* free the data using **deallocatePasteboardData:length:**.

See also: – **writeType:fromStream:**

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.

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

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

writeFileContents:

– (BOOL)**writeFileContents:**(const char *)*filename*

Writes the contents of the file *filename* to the pasteboard, and declares the data to be of type NXFileContentsPboardType and also of a type appropriate for the file's extention (as returned by **NXCreateFileContentsPboardType()** when passed the files extention), if it has one. Returns YES if the data from *filename* was successfully written to the pasteboard, and NO otherwise.

See also: – **readFileContentsType:toFile:**

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:**

writeType:fromStream:

– **writeType:**(const char *)*dataType* **fromStream:**(NXStream *)*stream*

Writes the type *dataType* to the pasteboard from the supplied stream *stream*. The stream must be readable. If the stream is seekable, it is seeked back to the beginning before the data is read; otherwise, data is read from the current position. In either case, all data to the end of the stream is read.

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: – **writeType:data:length:**

Method Implemented By The Owner

pasteboardChangedOwner:

– **pasteboardChangedOwner:***sender*

Notifies a prior owner of the *sender* Pasteboard (and owners of representations on the pasteboard) that the pasteboard has changed owners. This method is optional and need only be implemented by pasteboard owners that need to know when they have lost ownership. The owner is not able to read the contents of the pasteboard when responding to this method. The owner should be prepared to receive this method at any time, even from within the **declareTypes:num:owner:** used to declare ownership.

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:**

PopUpList

Inherits From: Menu : Panel : Window : Responder : Object

Declared In: appkit/PopUpList.h

Class Description

A PopUpList is a type of Menu that's used to make choices from a limited set of options, usually in a specific context. The PopUpList is usually triggered to pop up by a Button, and tracks the mouse like a Menu does until the user releases the mouse, at which time the PopUpList sends its action message to its target and disappears. Depending on the type of the PopUpList (see below), the title of the trigger Button is set to the title of the item selected from the PopUpList. Though a PopUpList is a user-interface device and sends an action message, it's not a Control class. The trigger that pops it up, however, is a Control; it's nearly always a Button, but may be a Cell in a Matrix, or a subclass of View that responds to the **setTitle:** and **title** messages.

There are actually two types of PopUpList: The pop-up list and the pull-down list. The type is set with the **changeButtonTitle:** method. A pop-up list's trigger always displays the item that was last selected, so a pop-up list is often used for selecting items from a small- to medium-sized set of options (like the zoom factor for a document Window). It's a useful alternative to a Matrix of radio Buttons or an NXBrowser when screen space is at a premium; a zoom factor pop-up can easily be fit next to a scroll bar on the bottom of a Window, for example. If there are very many items in the set of options, however, a pop-up list can nearly fill the height of the screen; in this case it would be better to use an NXBrowser, possibly in its own Panel.

A pull-down list is generally used for selecting actions in a very specific context, like the "Operations" pull-down list in Interface Builder's Classes browser. It has a "title" item that is always displayed on the trigger. When the actions only make sense in the context of a particular display, a pull-down list can be used in that display to keep the related actions nearby, and to keep them out of the way when that display isn't visible. This also helps reduce clutter in Menus.

Using PopUpLists with Interface Builder

Interface Builder contains a palette item that looks like a pop-up list. This item is actually a trigger Button for a PopUpList, which is the target of the Button. You can change the list to be a pull-down list with Interface Builder's Button Inspector, which shows radio buttons for selecting a pop-up or pull-down list instead of the usual Button options.

If you create an outlet from some other object and connect it to the graphical PopUpList as shown on the screen, you're actually connecting the outlet to the trigger Button. If you need a connection directly to the PopUpList, you should reset the outlet at run-time. This can be done in your Application delegate's **appDidInit:** method, in the **awake** method of the object containing the outlet, or preferably in the **awakeFromNib** method of the interface module's "File's Owner" class (**awakeFromNib** is described in the NXNibNotification protocol specification). For example, if the object has an outlet called **popup**:

```
- awakeFromNib
{
    [super awakeFromNib];
    if (![popup isKindOf:[PopUpList class]]) popup = [popup target];
    /* other setup code */
    return self;
}
```

If you need connections to both the trigger Button and the PopUpList itself, create outlets with names that distinguish the two, like **popupButton** and **popupList**, and only connect **popupButton** in Interface Builder. The object's **awakeFromNib** or other such method can then set the **popupList** outlet from **popupButton**'s target.

Creating a PopUpList Programmatically

To create a PopUpList programmatically, simply allocate an instance, send it an **init** message, and use the **changeButtonTitle:** method to configure the PopUpList as a pop-up or pull-down list. Your code can then add whatever items are needed with the **addItem:** method, or configure the PopUpList in other ways.

Once a PopUpList has been built, it must be attached to a trigger, which is usually a Button (though it may also be a ButtonCell). There are two functions that attach a PopUpList to a Button, as well as making the Button look like a pop-up or pull-down list by adding the appropriate icon and setting other parameters. **NXAttachPopUpList()** attaches the PopUpList to a Button or ButtonCell passed to the function.
NXCreatePopUpListButton() creates and returns a Button that triggers the PopUpList passed to it; your code can then add the Button to a View in your application.

Note: If you use **NXAttachPopUpList()** with a Button whose title doesn't appear in the PopUpList, the PopUpList will add that title to the top of its list when the Button triggers it. This is desirable for a pull-down list, since you won't have to add the title item to the list yourself, but it should be avoided for pop-up lists. Specifically, this is not a reliable means of adding items to the PopUpList; use **addItem:** for that.

Working with a PopUpList

PopUpList is actually a subclass of Menu that contains a Matrix of MenuCells (Menu's **itemList** method can be used to get the Matrix from the PopUpList). When the PopUpList's target is sent the action message, the sender of that message is actually the Matrix. The trigger Button itself can't be accessed by the target of the PopUpList. To get the PopUpList itself from the sending Matrix, the target can use View's **window** method (since PopUpList is a kind of Window). If the target need specific Cells, it can ask the Matrix directly for those with **selectedCell**, **cellAt::**, and other such methods.

If the title of a pull-down list needs to be changed, both the title of the trigger Button and the title item of the PopUpList itself need to be changed. The easiest way to do this is to change the Button's title, and to remove the title item from the pull-down list with **removeItemAt:** (it's kept at position 0). When the PopUpList is next triggered by the Button, it will add the Button's new title to the top of its list if that title isn't already in the list.

If you want to change the title of a pop-up list's trigger Button, be aware that this title represents the selected item to the user, so your code will have to also change the selected Cell in the PopUpList's Matrix. It can do this either by scanning for the title, or by changing the selected Cell by position first, then getting that Cell's title to use as the title of the trigger Button.

For more information, see the class specifications for Matrix, MenuCell, and Button

Instance Variables

None declared in this class.

Method Types

Initializing a PopUpList	– init
Setting up the items	– addItem: – insertItem:at: – removeItem: – removeItemAt: – indexOfItem: – count
Interacting with the trigger Button	– changeButtonTitle: – getButtonFrame:

Activating the PopUpList	– popUp:
Getting the user's selection	– selectedItem
Modifying the items	– setFont: – font
Target and action	– setAction: – action – setTarget: – target
Resizing the PopUpList	– sizeWindow::

Instance Methods

action

– (SEL)**action**

Returns the action sent to the PopUpList's target when an item is selected from the list. This is actually the action message of the PopUpList's Matrix.

See also: – **setAction:**, – **action** (Matrix), – **target**

addItem:

– **addItem:**(const char *)*title*

Adds an item with the name *title* to the bottom of the PopUpList, and returns the MenuCell created for that item (so a key equivalent can be added, for example). If an item with the name *title* already exists in the PopUpList, this method does nothing and returns **nil**.

See also: – **insertItem:at:**, – **removeItem:**

changeButtonTitle:

– **changeButtonTitle:**(BOOL)*flag*

If *flag* is YES, then when a selection is made from the list, the title of the selected item becomes the title on the Cell of the trigger that sent the **popUp:** message (nearly always a Button, but sometimes a Matrix). This makes the Button appear to the user as a pop-up list, with a small rectangular knob as the icon. If *flag* is NO, then the Button's title doesn't change, so that it appears to the user as a pull-down list, with a small inverted triangular mark for an icon. The default is YES (that is, a pop-up list). Returns **self**.

count

– (unsigned int)**count**

Returns the number of items in the PopUpList. If the PopUpList is configured as a pull-down list, this number includes the MenuCell that holds the pull-down list's title.

font

– **font**

Returns the Font used to draw the items in the PopUpList.

See also: – setFont:

getButtonFrame:

– **getButtonFrame:**(NXRect *)*bFrame*

Returns **self**, and by reference in *bFrame* the frame for the trigger that last popped up the PopUpList. The origin of the frame is set to (0.0, 0.0), so this method effectively returns the size of the trigger.

indexOfItem:

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

Returns the index of the item with the name *title*, or –1 if no such item is in the PopUpList.

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; your code should not use those methods with a 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 with the name *title* at position *index* in the PopUpList. The item with an index of 0 is the one at the top. Returns the newly inserted MenuCell.

If an item with a title of *title* already exists in the PopUpList, it's removed and the new one is added. This essentially moves *title* to a new position, though if the item removed was at a position before *index*, the new item will actually be inserted at *index* + 1. If you want to move an item, it's better to invoke **removeItem:** explicitly and then send **insertItem:at:**.

See also: – **addItem:**, – **removeItemAt:**

popUp:

– **popUp:***trigger*

Pops up the PopUpList over the location of *trigger*, after resizing the PopUpList to be as wide as *trigger*. This is the action method sent by the trigger object to the target PopUpList. If the mouse goes up in an item of the PopUpList, the Matrix that displays the PopUpList's entries sends the action message to the target. If the PopUpList is a pop-up type list (set with **changeButtonTitle:**), *trigger*'s title is set to the title of the selected item in the PopUpList; *trigger*'s icon is not altered by this method. Returns **self**.

This method works if and only if the following conditions are met. The Application object's current event must be a mouse-down, and that mouse-down must have occurred within *trigger*'s frame; this method can therefore be effectively invoked only as a result of a mouse-down occurring in *trigger*. *trigger* must also be either a subclass of View that responds to the messages **title** and **setTitle:**, or a subclass of Matrix whose selected Cell responds to **title** and **setTitle:**. If there are no items in the PopUpList and *trigger*'s title is NULL, this method does nothing.

If *trigger*'s title isn't in the PopUpList, it's added as an item at the top before the PopUpList pops up. The list pops up with the item having the same title as *trigger* (either a pop-up list's selected item or the "title" item of a pull-down list) exactly over *trigger* if possible; if this would cause part of the list to be off the top or bottom of the screen, the entire list is shifted up or down so that it can fit on screen.

If any of the MenuCells in the PopUpList's Matrix bring up submenus (that is, have a Menu as a target and **submenuAction:** as the action message), they are changed to simply be title-displaying MenuCells, and will never bring up their submenus. Essentially, this means that you can't create a hierarchical PopUpList with this class unless you completely override this method.

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

removeItem:

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

Removes and returns the MenuCell with the name *title*. If there is no such MenuCell, returns **nil**.

See also: – **removeItemAt:**

removeItemAt:

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

Removes the MenuCell for the item at the specified *index*. Returns the MenuCell at that location, or **nil** if there was no such MenuCell.

See also: – **removeItem:**

selectedItem

– (const char *)**selectedItem**

Returns the title of the item last selected by the user (the item that was highlighted when the user released the mouse button), or NULL if for some reason there is no selected item. It is possible for a pull-down list's selected item to be its title item.

The target of the PopUpList can get the title of the selected item in one of two ways. Since the sender of the action message is actually the PopUpList's Matrix, the target can ask the Matrix for its selected Cell, and then ask that Cell for its title. Also, the PopUpList is the Matrix's Window, so the target can retrieve that and then send **selectedItem** to the PopUpList. These two methods can be coded as follows:

```
item = [[sender selectedCell] title];  // sender is actually a Matrix
item = [[sender window] selectedItem]; // PopUpList is Matrix's Window
```

The first example is the preferred way to get the title.

See also: – **selectedCell:** (Matrix), – **title** (Cell)

setAction:

– **setAction:**(SEL)*aSelector*

Sets the action sent to the PopUpList's target when an item is selected. The action message is actually sent by the Matrix containing the MenuCells that make up the PopUpList. Returns **self**.

A pull-down list does send its action if the mouse goes up in its title item.

See also: – **action**, – **setAction:** (Matrix), – **setTarget:**

setFont:

– **setFont:***fontObject*

Sets the Font used to draw the PopUpList's items. The PopUpList does redraw itself, but since it normally won't be on the screen when it receives this message, this shouldn't cause any undesirable side-effects. Returns **self**.

See also: – **font** (Matrix)

setTarget:

– **setTarget:***anObject*

Sets the object to which an action will be sent when an item is selected from the PopUpList. The action is actually sent by the Matrix containing the MenuCells that make up the PopUpList. Returns **self**.

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

sizeWindow::

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

Your code should never invoke this method, though you're free to override it. 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. It simply adds 1.0 to each dimension and sends **sizeWindow::** to **super**. 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 up the responder chain. The target is actually sent the action by the PopUpList's Matrix.

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

PrintInfo

Inherits From: Object

Declared In: appkit/PrintInfo.h

Class Description

A PrintInfo object stores information that's used during printing. The Application object automatically creates a PrintInfo object that, by default, is used for all printing jobs (for that application). You can create any number of additional PrintInfo objects; however, only one can be "active" at a time, as set through Application's **setPrintInfo:** method. The currently active PrintInfo object is returned through Application's **printInfo** method.

Although you can set a PrintInfo's attributes through the methods it provides, this is usually the task of other objects, notably the PageLayout and PrintPanel objects. The View or Window that's being printed may also supercede some PrintInfo settings. In particular, a View or Window can supply the range of pages in the document and can provide its own pagination mechanism through the **knowsPagesFirst:last:** and **getRect:forPage:** methods (see the documentation of these methods in the View class for details).

If the printed View or Window doesn't supply a pagination, the PrintInfo's vertical and horizontal pagination constants are used to trigger built-in pagination mechanisms:

Pagination Constant	Meaning
NX_AUTOPAGINATION	The image is diced into equal-sized rectangles and placed in one column of pages.
NX_FITPAGINATION	The image is scaled to produce one column (horizontal) or one row (vertical) of pages.
NX_CLIPPAGINATION	The image is clipped to produce one column or row of pages.

Vertical and horizontal pagination needn't be the same. However, if either dimension is scaled (NX_FITPAGINATION), the other dimension is scaled by the same amount to avoid stretching the image. If both dimensions are scaled, the scaling factor that produces the smallest image is used. Note that PrintInfo's scaling factor (as set through **setScalingFactor:**) is independent of the scaling that's imposed by pagination and is applied after the document has been paginated.

The PrintInfo attributes that describe a size on a sheet of paper (the margins and the size of the paper rectangle) are in points, where 72 points equals one inch.

Page numbers (**firstPage**, **lastPage**, and so on) are as they appear in the document. For example, to print the first three pages of a document that contains pages numbered from 20 to 29, **firstPage** would be set to 20 and **lastPage** to 22.

Instance Variables

```
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  allPages:1;
        unsigned int  horizPagination:2;
        unsigned int  vertPagination:2;
        unsigned int   printerIsOld:1;
        unsigned int   reversePageOrder:1;
} pInfoFlags;
int firstPage;
int lastPage;
int currentPage;
int copies;
char *outputFile;
DPSContext context;
short pagesPerSheet;
NXPrinter *printerObject;
id jobFeaturesTable;
const char *paperFeed;
```

paperType	Type of paper.
paperRect	Rectangle representing the paper's area.
leftPageMargin	Size of the left margin in points.
rightPageMargin	Size of the right margin in points.
topPageMargin	Size of the top margin in points.
bottomPageMargin	Size of the bottom margin in points.
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.allPages	True if all the pages are to be printed.
pInfoFlags.horizPagination	Horizontal pagination mode.
pInfoFlags.vertPagination	Vertical pagination mode.
firstPage	Page number of the first page to print.
lastPage	Page number of the last page to print.
currentPage	Page number of the page currently being printed.
copies	Number of copies to print.
outputFile	File to spool to.
context	Spooling context.
pagesPerSheet	The number of pages per sheet of paper.
printerObject	The printer that the printing job will run on.
jobFeaturesTable	Table of additional printing job attributes.
paperFeed	Paper feed slot name.

Method Types

Initializing and freeing a PrintInfo instance
- init
- free

Defining the printing rectangle
- setMarginLeft:right:top:bottom:
- getMarginLeft:right:top:bottom:
- setOrientation:andAdjust:
- orientation
- setPaperRect:andAdjust:
- paperRect
- setPaperType:andAdjust:
- paperType

Page range
- setFirstPage:
- firstPage
- setLastPage:
- lastPage
- setAllPages:
- isAllPages
- currentPage

Pagination and scaling
- setHorizPagination:
- horizPagination
- setVertPagination:
- vertPagination
- setScalingFactor:
- scalingFactor

Positioning the image on the page
- setHorizCentered:
- isHorizCentered
- setVertCentered:
- isVertCentered
- setPagesPerSheet:
- pagesPerSheet

Print job attributes	– initializeJobDefaults – setJobFeature:toValue: – valueForJobFeature: – removeJobFeature: – jobFeatures – setPageOrder: – pageOrder – setReversePageOrder: – reversePageOrder – setCopies: – copies – setPaperFeed: – paperFeed
Specifying the printer	+ setDefaultPrinter: + getDefaultPrinter – setPrinter: – printer
Spooling	– setOutputFile: – outputFile – setContext: – context
Archiving	– read: – write:

Class Methods

getDefaultPrinter

+ (NXPrinter *)**getDefaultPrinter**

Returns an NXPrinter object that corresponds to the user's default printer, as declared in the defaults database. If the printer can't be found, **nil** is returned.

See also: + **setDefaultPrinter:**, – **setPrinter:**, – **printer**

setDefaultPrinter:

+ **setDefaultPrinter:**(NXPrinter *)*printer*

Sets the user's default printer by writing the name and host of *printer* to the defaults database. Unless a PrintInfo's printer is otherwise set (through **setPrinter:**) the default printer is used for printing.

See also: + getDefaultPrinter, – setPrinter:, – printer

Instance Methods

context

– (DPSContext)**context**

Returns the Display PostScript context used for printing.

copies

– (int)**copies**

Returns the number of copies that will be printed.

currentPage

– (int)**currentPage**

Returns the page number of the page currently being printed. This method is valid only while printing or faxing.

firstPage

– (int)**firstPage**

Returns the page number of the first page that will be printed, as set through **setFirstPage:**, or MININT if the value hasn't been explicitly set. If all pages are being printed, the first page value is ignored during printing. If the page order is reversed, this value gives the page number of the last page that will be printed.

See also: – setFirstPage:, – setAllPages:, – setPageOrder:

free

– **free**

Frees the PrintInfo object.

getMarginLeft:right:top:bottom:

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

Returns, by reference, the sizes of the four page margins measured in points.

See also: – **setMarginLeft:right:top:bottom:**

horizPagination

– (int)**horizPagination**

Returns a constant that represents the manner in which an image is distributed horizontally among pages. See the class description, above, for a description of the pagination constants.

See also: – **setHorizPagination:**

init

– **init**

Initializes the PrintInfo object after memory for it has been allocated through **alloc** or **allocFromZone:**. Returns **self**.

initializeJobDefaults

– **initializeJobDefaults**

Called before each print job (specifically, before the Print panel is displayed), this method initializes the following PrintInfo attributes:

Attribute	Value
First page	MININT
Last page	MAXINT
Copies	1
Page order	First-to-last
Printer	The user's default printer
Paper feed	The default paper feed slot

isAllPages

– (BOOL)**isAllPages**

Returns whether all pages will be printed. If NO, only those pages that fall within [**firstPage**, **lastPage**] will be printed.

See also: – **setAllPages:**

isHorizCentered

– (BOOL)**isHorizCentered**

Returns whether the image is centered horizontally on a page; if this returns NO, the image is flush against the left margin. If the image spills over more than one page horizontally, this value is ignored and the image is always set against the left margin.

isVertCentered

– (BOOL)**isVertCentered**

Returns whether the image is centered vertically on a page; if this returns NO, the image is flush against the top margin. If the image spills over more than one page vertically, then this value is ignored and the image is always set against the top margin.

See also: – **setVertCentered:**

jobFeatures

– (const char **)**jobFeatures**

Returns a pointer to an array of pointers that contains the keys to the job features table, the hash table that contains additional printing-job attributes. You would use these keys as arguments to methods such as **valueForJobFeature:** and **removeJobFeature:**. It's the caller's responsibility to free the pointer to the array, but not the pointers in the array.

See also: – **setJobFeature:toValue:**, – **valueForJobFeature:**, – **removeJobFeature:**

lastPage

– (int)**lastPage**

Returns the page number of the last page that will be printed, as set through **setLastPage:**, or MAXINT if the value hasn't been explicitly set. If all pages are being printed, the last page value is ignored during printing. If the page order is reversed, this value gives the page number of the first page that will be printed.

See also: – **setLastPage:**, – **setAllPages:**, – **setReversePageOrder:**

orientation

– (char)**orientation**

Returns the **page orientation** as NX_PORTRAIT or NX_LANDSCAPE.

See also: – **setOrientation:andAdjust:**, – **setPaperType:andAdjust:**, – **setPaperRect:andAdjust:**

outputFile

– (const char *)**outputFile**

Returns the name of the file to which the generated PostScript code is sent. If this is NULL, the code is written to a temporary file.

See also: – **setOutputFile:**

pageOrder

– (char)**pageOrder**

Returns a constant that denotes the order in which pages are printed. See **setPageOrder:** for the page order constants.

See also: – **setPageOrder:**

pagesPerSheet

– (short)**pagesPerSheet**

Returns the number of pages per sheet of paper.

paperFeed

– (const char *)**paperFeed**

Returns the name of the currently used paper feed slot.

See also: – **setPaperFeed:**

paperRect

– (const NXRect *)**paperRect**

Returns a pointer to a rectangle that gives the size of the paper, measured in points. Note that the rectangle is useful only for its size field; the origin of the paper is always (0.0, 0.0).

paperType

– (const char *)**paperType**

Returns the paper type. If the type is unknown, an empty string is returned.

printer

– (NXPrinter *)**printer**

Returns the NXPrinter that's used for printing.

read:

– **read:**(NXTypedStream *)*stream*

Reads the PrintInfo from the typed stream *stream*.

removeJobFeature:

– **removeJobFeature:**(const char *)*key*

Removes, from the job-features hash table, the element that corresponds to *key*.

See also: – **jobFeatures**, – **setJobFeature:toValue:**, – **valueForJobFeature:**

reversePageOrder

– (BOOL)**reversePageOrder**

Returns YES if the PrintInfo dictates that pages will be printed in reverse order (in other words, if the ordering established through **setPageOrder:** is reversed).

See also: – **setReversePageOrder:**, – **setPageOrder:**

scalingFactor

– (float)**scalingFactor**

Returns the factor by which the image is scaled.

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 Display PostScript context used for printing. This is normally done by the printing machinery in View.

setCopies:

– **setCopies:**(int)*anInt*

Sets the number of copies that will be printed.

setFirstPage:

– **setFirstPage:**(int)*anInt*

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

setHorizCentered:

– **setHorizCentered:**(BOOL)*flag*

Sets whether the image is centered horizontally on a page; if flag is NO, the image is flush against the left margin. If the image spills over more than one page horizontally, then *flag* is ignored and the image is always against the left margin.

setHorizPagination:

– **setHorizPagination:**(int)*mode*

Sets the way in which a document is divided horizontally into pages. See the class description, above, for the pagination constants that you can use as the argument to this method.

setJobFeature:toValue:

– **setJobFeature:**(const char *)*feature* **toValue:**(const char *)*value*

Sets the value of the given job feature. The feature is added to the job-features hash table if it isn't already present.

setLastPage:

– **setLastPage:**(int)*anInt*

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

setMarginLeft:right:top:bottom:

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

Sets the margins. All margins are in points.

setOrientation:andAdjust:

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

Sets the orientation or the page; *mode* should be either NX_PORTRAIT or NX_LANDSCAPE.

If *flag* is NO, then only the orientation is affected. If *flag* is YES, the paper rectangle value is 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 *aString* is NULL, the code is sent to a temporary file.

setPageOrder:

– **setPageOrder:**(char)*mode*

Sets the order in which pages are printed as one of these constants:

NX_DESCENDINGORDER
NX_SPECIALORDER
NX_ASCENDINGORDER
NX_UNKNOWNORDER

See also: – reversePageOrder, – pageOrder

setPagesPerSheet:

– **setPagesPerSheet:**(short)*pageCount*

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

setPaperFeed:

– **setPaperFeed:**(const char *)*paperFeedSlot*

Sets the paper feed slot by name. If *paperFeedSlot* is NULL (or an empty string), any paper feed slot is acceptable; to choose the manual feed slot, set the name to "NXManual". Any other name that you use must appear in the PPD table (as documented in the NXPrinter class). Returns **self**.

See also: – **setPaperFeed:**

setPaperRect:andAdjust:

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

Sets the size of the paper, measured in points, that's to be used in printing. The origin of the rectangle is always set to (0,0)—the origin field of *aRect* is ignored.

If *flag* is NO, then only the paper rectangle is changed. If *flag* is YES, the orientation and paper type values are updated to reflect the new size.

setPaperType:andAdjust:

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

Sets the name of the paper type.

If *flag* is NO, only the paper type is changed. If *flag* is YES, the paper rectangle and orientation values are updated to reflect the new type (given that *type* is a recognized paper type).

setPrinter:

– **setPrinter:**(NXPrinter *)*aPrinter*

Sets the printer that's used in subsequent printing jobs.

setReversePageOrder:

– **setReversePageOrder:**(BOOL)*flag*

Sets whether pages are printed in reverse order. This ordering is applied to the page order mode set through **setPageOrder:**. Returns **self**.

See also: – **reversePageOrder**, – **setPageOrder:**

setScalingFactor:

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

Sets the amount by which the document is scaled. Note that the scaling you set has no effect if the document does its own pagination through View's **knowsPagesFirst:last:** method.

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 a document is divided vertically into pages. See the class description, above, for the pagination constants that you can use as the argument to this method.

valueForJobFeature:

– (const char *)**valueForJobFeature:**(const char *)*feature*

Returns the value for the given printing feature, as stored in the job-features table.

See also: – **jobFeatures**, – **setJobFeature:toValue:**, – **removeJobFeature**

vertPagination

– (int)**vertPagination**

Returns a constant that represents the manner in which an image is distributed vertically among pages. See the class description, above, for a description of the pagination constants.

See also: – **setVertPagination:**

write:

– **write:**(NXTypedStream *)*stream*

Writes the PrintInfo to the typed stream *stream*.

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 a print job, such as which pages and how many copies to print. The PrintPanel is created, displayed, and run (in a modal loop) when a **printPSCode:** message is sent to a View or Window, provided that the sender of the message doesn't implement the **shouldRunPrintPanel:** method to return NO (if the method isn't implemented, or if it returns YES, the Panel is brought up).

Each application can have but one PrintPanel object. If you're creating a subclass of PrintPanel, you should send your subclass' class object a **new** message (without invoking **alloc** or **allocFromZone:**) before any **printPSCode:** messages are sent to ensure that an instance of your subclass is the unique PrintPanel object for your application.

Short of subclassing PrintPanel, you can augment it's display by adding a custom View through the **setAccessoryView:** method. The Print panel is automatically resized to accommodate the View that you add. Note, however, that you don't have to create controls for special printer features. If a printer includes features in the "OpenUI" field of its PostScript Printer Description (PPD) table, these features will be displayed in a separate panel that's brought up when the user clicks the Print panel's Options button.

The Print panel lists the names of the printers that are available for printing. If only one printer is available, the list isn't displayed.

Instance Variables

id **appIcon**;
id **pageMode**;
id **firstPage**;
id **lastPage**;
id **copies**;
id **ok**;
id **cancel**;
id **preview**;
id **save**;
id **printers**;
id **feed**;
id **resolutionList**;
id **name**;
id **note**;
id **status**;
int **exitTag**;
id **accessoryView**;
id **buttons**;
id **optionsButtons**;

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.
printers	The Matrix of available printers.
feed	The PopUpList of paper feed options.
resolutionList	The PopUpList of resolution choices.

name	The TextField for the name of the currently chosen printer.
note	The TextField for the note for the currently chosen printer.
status	The TextField for the status of the requested operation.
exitTag	The tag of the button the user clicked to exit the panel.
accessoryView	The optional View added by the application.
buttons	The Matrix of Print panel buttons.
optionsButton	The Matrix of Options panel 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
– pickedAllPages:
– textWillChange:

Communicating with the PrintInfo object
– updateFromPrintInfo
– finalWritePrintInfo

Class Methods

alloc

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

allocFromZone:

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

new

+ **new**

Returns the application's sole PrintPanel object, creating it if necessary. You rarely need to invoke this method directly; the PrintPanel object is created automatically when **printPSCode:** is sent to a View or Window.

newContent:style:backing:buttonMask:defer:

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

Initializes the PrintPanel object. You never invoke this method; use **new** instead.

Instance Methods

accessoryView

– **accessoryView**

Returns the PrintPanel's accessory view.

See also: – **setAccessoryView:**

finalWritePrintInfo

– **finalWritePrintInfo**

Writes the values of the PrintPanel's printing attributes to the application's PrintInfo object.

See also: – **updateFromPrintInfo**

free

– **free**

Frees the PrintPanel object and its contents, including the accessory view.

pickedAllPages:

– **pickedAllPages:***sender*

The target of the Pages radio buttons, this method updates the text that's displayed in the From and To (pages) fields. By default, the fields are empty if the All radio button is clicked and set to "first" and "last" if the From/To button is clicked. These strings can only be changed by overriding this method.

pickedButton:

– **pickedButton:***sender*

The target of the Print, Cancel, Preview, Save, and Fax buttons, this method ends the Print panel's modal run. If a button other than Cancel inspired this method, then the Print panel's Copies field and the From and To (pages) fields must be acceptable (they must hold positive numbers), otherwise the unacceptable entry is selected and the panel isn't stopped. If the panel is being cancelled, it's stopped regardless of the entries' acceptability. Upon successful dismissal, the **exitTag** instance variable is set to the tag of the button that the user clicked to dismiss the panel (NX_OKTAG, NX_CANCELTAG, NX_PREVIEWTAG, NX_SAVETAG, or NX_FAXTAG).

runModal

– (int)**runModal**

Reads the pertinent data from the PrintInfo object into the PrintPanel object and then runs the Print panel in a modal loop. When the user clicks one of the buttons at the bottom of the panel (Print, Cancel, Preview, Save, or Fax), the modal loop is broken (by the **pickedButton:** method) and, if user didn't cancel the panel, the new Print information is written to the PrintInfo object. This method returns the tag of the button that the user clicked to dismiss the panel (NX_OKTAG, NX_CANCELTAG, NX_PREVIEWTAG, NX_SAVETAG, or NX_FAXTAG).

This method is normally invoked from Window or View's **printPSCode:** method. You should note that the feat that's requested by the user (printing, faxing, etc.) is performed by **printPSCode:**. Furthermore, this method doesn't hide the Print panel after the modal loop is broken; that, too, is left to **printPSCode:**.

setAccessoryView:

– **setAccessoryView:***aView*

Adds *aView* to the PrintPanel's view hierarchy. Applications can invoke this method to add a View that contains their own controls. The 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 the panel's current accessory view, if any, is removed. Returns the old accessory View.

See also: – **accessoryView**

textWillChange:

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

Inovoked when the user types in the From or To (pages) fields, this method ensures that the correct Pages radio button is selected (From/To rather than All). If the To value is greater than the From value, the PrintInfo object will be set to print pages in reverse order.

updateFromPrintInfo

– **updateFromPrintInfo**

Reads the application's PrintInfo object, setting the initial values of this panel. This method is invoked automatically when the PrintPanel receives a **runModal** message.

See also: – **finalWritePrintInfo**

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 methods and instance variable that pass event and action messages along the responder chain.

Instance Variables

id **nextResponder**;

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

Method Types

Freeing an instance	– free
Setting 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:
Help menu support	– helpRequested:
Archiving	– read: – write:

Instance Methods

acceptsFirstResponder

– (BOOL)**acceptsFirstResponder**

Returns NO to indicate that, by default, a Responder doesn'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.

free

– **free**

Frees the space used by a Responder instance and removes it from the hash table used to locate help. Returns **self**.

helpRequested:

– **helpRequested:**(NXEvent *)*eventPtr*

Invoked by a Window instance when the user has clicked for help. The Window instance sends this message to the first responder. The receiver shows its help panel if it has one, and if not forwards the message to the next responder. If there is no next responder to respond, the method executes **NXBeep()**. Your application should never invoke this method directly. Returns **self**.

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*

Responds to an event message that has reached the end of the responder chain without finding an object that can respond. When the event is a key down, **noResponderFor:** 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 **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*. Returns **self**.

See also: – **write:**

resignFirstResponder

– **resignFirstResponder**

Notifies the receiver that it has been asked to relinquish its status as 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 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 untargeted 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:** (Application),
– **readSelectionFromPasteboard:** (Application)

write:

– **write:**(NXTypedStream *)*stream*

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

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 file name to have a certain file type, as specified by a file name 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 **panelValidateFilenames:** is 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 valid, 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

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	The form in which the user types file names
browser	The browser displaying the file hierarchy
okButton	The OK button
accessoryView	Application-customized area
separator	The line separating the icon from the rest of the panel
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	Specifies whether file is being opened or saved
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	Length of the directory string

Method Types

Creating and freeing a SavePanel	+ newContent:style:backing:buttonMask:defer: – free
Setting the SavePanel class	+ setSavePanelFactory:
Customizing the SavePanel	– setAccessoryView: – accessoryView – setTitle: – setPrompt:
Setting directory and file type	– setDirectory: – setRequiredFileType: – requiredFileType
Handling file packages	– doesTreatFilePackagesAsDirectories – setTreatsFilePackagesAsDirectories:
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

alloc

+ 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:

+ **allocFromZone:**(NXZone *)*zone*

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:**

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.

See also: + **setSavePanelFactory:**

setSavePanelFactory:

+ **setSavePanelFactory:***class*

Sets the class from which SavePanel will be instantiated. *class* should be a subclass of SavePanel. If you want to use your own SavePanel subclass, your application must invoke this method before it creates the shared instance of SavePanel, or else free the shared instance before invoking this method. When the **newContent:style:backing:buttonMask:defer:** method is invoked, the object it returns will belong to *class*.

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

Instance Methods

accessoryView

– **accessoryView**

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

See also: – **setAccessoryView:**

cancel:

– **cancel:***sender*

This method is the action message sent by the Cancel button to 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.

doesTreatFilePackagesAsDirectories

– (BOOL)**doesTreatFilePackagesAsDirectories**

Tests whether the SavePanel displays file packages to the user as directories. Returns YES (the default) if the user is shown files and subdirectories within a file package. Returns NO if the user is shown only file package names, with no indication that they are directories.

See also: – **setTreatsFilePackagesAsDirectories:**

filename

– (const char *)**filename**

Returns the filename—including the path to the file—that the SavePanel last accepted.

free

– **free**

Frees all storage used by the SavePanel.

ok:

– **ok:***sender*

This method is the action message sent by the OK button to the SavePanel.

requiredFileType

– (const char *)**requiredFileType**

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

runModal

– (int)**runModal**

Displays the panel and begins its event loop. Invokes Application's **runModalFor:** method with **self** as the argument. Returns the constant returned by that method, depending on the method used to stop the modal event loop.

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

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 modal event loop. Invokes Application's **runModalFor:** method with **self** as the argument. Returns the constant returned by that method, depending on the method used to stop the modal event loop.

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

selectText:

– **selectText:***sender*

Advances the current browser selection one line when Tab is pressed (goes back one line when Shift-Tab 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 method can be called repeatedly to change the accessory view depending on the situation. If *aView* is **nil**, any accessory view 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 into 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 that don't already have that extension; for example, "nib". *type* should not include the period that begins the extension. Be careful to invoke this method each time the SavePanel is used for another file type within the application. Returns **self**.

setTreatsFilePackagesAsDirectories:

–**setTreatsFilePackagesAsDirectories:**(BOOL)*flag*

Sets the SavePanel's behavior for displaying file packages to the user. If *flag* is YES, the user is shown files and subdirectories within a file package. If NO, the user is shown only file package names, with no indication that they are directories.

See also: – doesTreatFilePackagesAsDirectories

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:compareFilenames::checkCase:

– (int)**panel:***sender*
compareFilenames:(const char *)*fileName1* **:**(const char *)*fileName2*
checkCase:(BOOL)*flag*

Controls the ordering of files presented by the SavePanel. This method should return 1 if *fileName1* should precede *fileName2*, 0 if the two names are equivalent, and -1 if *fileName2* should precede *fileName1*.

Don't reorder file names in the SavePanel lightly, since it may confuse the user to have files in one SavePanel or OpenPanel ordered differently than those in other such panels or in the Workspace Manager. SavePanel and OpenPanel's default behavior is to order files as they are in the Workspace Manager file viewer. Note also that implementing this method will reduce the operating performance of the panel.

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 valid, 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 the current file name in the SavePanel can be used; it returns YES if the file name is valid, or NO if the SavePanel should stay up and wait for the user to type in a different file name. The delegate can get the file name in the SavePanel by sending a **filename** message back to *sender*.

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 some 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 message that will be sent to the target 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

```
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 perform 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**.

See also: – **read**

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.

See also: – **setFloatValue::**

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 is causing the current action, typically the part 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 newly allocated 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)

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:**, – **awake**

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::**, – **floatValue**

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:**, – **floatValue**

setTarget:

– **setTarget:***anObject*

Sets the target of the Scroller to *anObject*. 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:**

ScrollView

Inherits From: View : Responder : Object

Declared In: appkit/ScrollView.h

Class Description

A ScrollView object lets the user interact with a document that's too large to be shown 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. If Scrollers are required, the application must send **setVertScrollerRequired:**YES and **setHorizScrollerRequired:**YES messages to the ScrollView; the content view is resized to fill the area of the ScrollView not occupied by the Scrollers.

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

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 remaining in the content view after a page scroll.
lineAmount	The number of units 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:...**, – **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 calculated 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:...**, – **getContentSize:**

Instance Methods

backgroundColor

– (NXColor)**backgroundColor**

Returns the color of the content view's background.

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

backgroundGray

– (float)**backgroundGray**

Returns the gray value of the content view's background.

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

borderType

– (int)**borderType**

Returns a value that represents the type of border surrounding the ScrollView, one of NX_NOBORDER, NX_LINE, NX_BEZEL, or NX_GROOVE.

See also: – **setBorderType:**

docView

– **docView**

Returns the current document view.

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

drawSelf::

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

Draws the ScrollView, but not its subviews. 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 visible portion of the document view by forwarding this message to the content view. Returns the value returned by the forwarded message.

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

horizScroller

– **horizScroller**

Returns the horizontal scroller, a Scroller object.

See also: – **vertScroller**

initFrame:

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

Initializes the ScrollView. 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 the content view. Clipping is turned off (the ScrollView relies on the content view for clipping), opacity is set to YES, and autoresizing is set to YES. 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: – **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*

Moves, resizes, or disables the Scrollers when the document is autoscrolled or changes size. Returns **self**.

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. Returns the content view.

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. Returns the content view.

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. This method simply invokes (and returns the value returned by) the content view's **setCopyOnScroll:** method.

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

setDisplayOnScroll:

– **setDisplayOnScroll:**(BOOL)*flag*

Determines whether the results of scrolling will be immediately displayed. This method simply invokes (and returns the value returned by) the content view's **setDisplayOnScroll:** method.

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

setDocCursor:

– **setDocCursor:***anObj*

Sets the cursor to be used inside the content view. Returns the old cursor.

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

setDocView:

– **setDocView:***aView*

Attaches the document view. 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 *anObject* (which should be an instance of a subclass of Scroller). 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 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 shrinks its other subviews to accommodate it. 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 doesn't 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 *anObject* (which should be an instance of a subclass of Scroller). 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 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 shrinks its other subviews to accommodate it. 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 doesn't 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**

Determines the layout of the ScrollView by setting the sizes and locations of the object's subviews. You rarely send a **tile** message directly; you may override it if you need to have the ScrollView manage additional views. A **tile** message is sent whenever the ScrollView is resized, or a vertical or horizontal scroller is added or removed. This method *doesn't* redisplay the ScrollView. Returns **self**.

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

vertScroller

– **vertScroller**

Returns the vertical scroller, a Scroller object.

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

None declared in this class.

Method Types

Initializing a SelectionCell	– init – initTextCell:
Determining component sizes	– calcCellSize:inRect:
Accessing graphic attributes	– setLeaf: – isLeaf – isOpaque
Displaying	– drawSelf:inView: – drawInside:inView: – highlight:inView:lit:
Archiving	– awake

Instance Methods

awake

– **awake**

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

See also: – **read:** (Cell)

calcCellSize:inRect:

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

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

drawInside:inView:

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

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

See also: – **drawSelf:inView:**, – **lockFocus** (View)

drawSelf:inView:

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

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

See also: – **drawInside:inView:**

highlight:inView:lit:

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

Sets the SelectionCell's highlighted state to *flag* and redraws it within *cellFrame* in *aView*. Returns **self**.

See also: – **highlight:inView:lit:** (Cell)

init

– **init**

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

See also: – **initTextCell:**, – **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: – **init**, – **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 the one that MenuCell displays to indicate submenus.

See also: – **setLeaf:**

isOpaque

– (BOOL)**isOpaque**

Returns YES, since SelectionCells draw over all the pixels in their frames.

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 display a right arrow like that displayed by MenuCell to indicate submenu entries. This method does not display the SelectionCell, even if autodisplay is on. Returns **self**.

See also: – **isLeaf:**

Slider

Inherits From: Control : View : Responder : Object

Declared In: appkit/Slider.h

Class Description

Slider is a type of Control with a sliding knob that can be moved to represent a value between a minimum and a maximum. A Slider may be either horizontal or vertical, but its minimum value is always at the left or bottom end of the bar, and the maximum at the right or top. By default, a Slider is a continuous Control: it sends it action message to its target continuously while the user drags its knob. To configure a Slider to send its action only when the mouse is released, send **setContinuous:** with an argument of NO.

A Slider can be configured to display an image, a title, or both, in its bar behind its knob. A Slider's title can be drawn in any gray level or color, and in any font available. A Slider's value can be set programmatically with any of the standard Control value-setting methods, such as **setFloatValue:**. For more information on the behavior of these methods in Slider, see the SliderCell class specification.

Instance Variables

None declared in this class.

Method Types

Setting Slider's Cell Class	+ setCellClass:
Initializing a new Slider	– initFrame:

Modifying a Slider's appearance	– setKnobThickness: – knobThickness – setImage: – image – setTitle: – setTitleNoCopy: – title – setTitleCell: – titleCell – setTitleFont: – titleFont – setTitleColor: – titleColor – setTitleGray: – titleGray – isVertical
Setting value limits	– setMinValue: – minValue – setMaxValue: – maxValue
Resizing the Slider	– sizeToFit
Handling events	– acceptsFirstMouse – setEnabled: – mouseDown:

Class Methods

setCellClass:

+ **setCellClass:***classId*

Configures the Slider class to use instances of *classId* for its Cells. *classId* should be the **id** of a subclass of SliderCell, obtained by sending the **class** message to either the SliderCell subclass object or to an instance of that subclass. The default Cell class is SliderCell. Returns **self**.

If this method isn't overridden by a subclass of Slider, then when it's sent to that subclass, Slider and any other subclasses of Slider that don't override the methods mentioned below will use the new Cell subclass as well. To safely set a Cell class for your subclass of Slider, override this method to store the Cell class in a static **id**. Also, override the designated initializer to replace the Slider subclass instance's Cell with an instance of the Cell subclass stored in that static **id**. See "Creating New Controls" in the Control class specification for more information.

Instance Methods

acceptsFirstMouse

– (BOOL)**acceptsFirstMouse**

Returns YES since Sliders always accept a mouse-down event that activates a Window, whether or not the Slider is enabled.

image

– **image**

Returns the NXImage that the Slider displays in its bar, or **nil** if one hasn't been set.

See also: – **setImage:**

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.

See also: – **sizeToFit**, – **isVertical**, – **isContinuous** (Control),
– **setContinuous:** (Control)

isVertical

– (int)**isVertical**

Returns 1 if the Slider is vertical, 0 if it's horizontal, and –1 if the orientation can't be determined (because the Slider hasn't been initialized, for example). A Slider is vertical if its height is greater than its width.

knobThickness

– (NXCoord)**knobThickness**

Returns the thickness of the Slider's knob (that is, its extent along the bar's length) in the Slider's coordinate system.

See also: – **setKnobThickness:**

maxValue

– (double)**maxValue**

Returns the maximum value of the Slider.

See also: – **setMaxValue:**, – **minValue**

minValue

– (double)**minValue**

Returns the minimum value of the Slider.

See also: – **setMinValue:**, – **maxValue**

mouseDown:

– **mouseDown:**(NXEvent *)*theEvent*

Tracks the mouse until a mouse-up event occurs, updating the knob's position to follow the cursor as it's dragged. Returns **self**.

See also: – **trackMouse:inRect:ofView:** (SliderCell)

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 changes. Returns **self**.

See also: – **isEnabled** (Control), – **setAutodisplay:** (View)

setImage:

– **setImage:***image*

Sets the NXImage used as the Slider's bar. This method doesn't scale the NXImage. Returns **self**.

setKnobThickness:

– **setKnobThickness:**(NXCoord)*aFloat*

Sets the thickness of the Slider's knob (that is, its extent along the bar's length) in its own coordinate system. *aFloat* must be greater than 0.0, and shouldn't be greater than the Slider's length. If the knob thickness changes, the Slider's inside is redrawn. Returns **self**.

See also: – **knobThickness**

setMaxValue:

– **setMaxValue:**(double)*aDouble*

Sets the maximum value of the Slider to *aDouble* and returns **self**. If the maximum value changes, the Slider's inside is redrawn to reposition the knob relative to the new maximum.

See also: – **maxValue**, – **setMinValue:**

setMinValue:

– **setMinValue:**(double)*aDouble*

Sets the minimum value of the Slider to *aDouble* and returns **self**. If the minimum value changes, the Slider's inside is redrawn to reposition the knob relative to the new minimum.

See also: – **minValue**, – **setMaxValue:**

setTitle:

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

Sets the title drawn over the Slider's bar to *aString*. Returns **self**.

See also: – **setTitleNoCopy:**, – **title**

setTitleCell:

– **setTitleCell:***aCell*

Sets the Cell used to draw the Slider's title. *aCell* should be an instance of TextFieldCell (or of a subclass). Doesn't redraw the Slider; furthermore, a **setTitle:** message is required to display a title, even if *aCell* already has a string value. Returns the old Cell.

See also: – **titleCell**, – **setTitle:**

setTitleColor:

– **setTitleColor:**(NXColor)*color*

Sets the color used to draw the Slider's title, redraws the Slider's inside, and returns **self**. The default is to draw in a gray level of 0.0 (NX_BLACK).

See also: – **titleColor**, – **setTitleGray:**, – **titleGray**

setTitleFont:

– **setTitleFont:***fontObject*

Sets the Font used to draw the SliderCell's title and redraws the Slider's inside. The default font is the default system font as set by the user (with the Preferences application), and its size is 12.0 point. Returns **self**.

See also: – **titleFont**

setTitleGray:

– **setTitleGray:**(float)*aFloat*

Sets the gray value used to draw the Slider's title, redraws the Slider's inside, and returns **self**. The default gray level is 0.0 (NX_BLACK).

See also: – **titleGray**, – **setTitleColor:**, – **titleColor**

setTitleNoCopy:

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

Sets the title drawn over the Slider's bar to *aString*, but doesn't copy the string. Returns **self**.

See also: – **setTitle:**, – **title**

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**.

title

– (const char *)**title**

Returns the string used as the Slider's title. The title is drawn over the Slider's bar. Returns **self**.

See also: – **setTitle:**

titleCell

– **titleCell**

Returns the TextFieldCell used to draw the Slider's title. If the Slider doesn't have a title, a new TextFieldCell is created and returned. This doesn't result in a title getting set.

See also: – **setTitleCell:**

titleColor

– (NXColor)**titleColor**

Returns the color used to draw the Slider's title. The default is to draw in a gray level of 0.0 (NX_BLACK). Returns **self**.

See also: – **setTitleColor:**, – **titleGray**, – **setTitleGray:**

titleFont

– **titleFont**

Returns the Font used to draw the Slider's title. The default font is the default system font as set by the user (with the Preferences application), and its size is 12.0 point.

See also: – **setTitleFont:**

titleGray

– (float)**titleGray**

Returns the gray value used to draw the Slider's title. The default gray level is 0.0 (NX_BLACK). Returns **self**.

See also: – **setTitleGray:**, – **titleColor**, – **setTitleColor:**

SliderCell

Inherits From: ActionCell : Cell : Object

Declared In: appkit/SliderCell.h

Class Description

SliderCell is a type of Cell used to assist the Slider class, and to build Matrices of sliders. See the Slider class specification for an overview of how SliderCells work.

Instance Variables

double **value**;
double **maxValue**;
double **minValue**;
NXRect **trackRect**;

value	The current value of the SliderCell.
maxValue	The maximum value of the SliderCell.
minValue	The minimum value of the SliderCell.
trackRect	The tracking area of the SliderCell, inside the bezel.

Method Types

Initializing a new SliderCell	– init
Determining component sizes	– calcCellSize:inRect: – getKnobRect:flipped:
Setting value limits	– setMinValue: – minValue – setMaxValue: – maxValue

Setting values	– setDoubleValue: – doubleValue – setFloatValue: – floatValue – setIntValue: – intValue – setStringValue: – stringValue
Modifying a SliderCell's appearance	– setKnobThickness: – knobThickness – setImage: – image – setTitle: – setTitleNoCopy: – title – setTitleCell: – titleCell – setTitleFont: – titleFont – setTitleColor: – titleColor – setTitleGray: – titleGray – isOpaque – isVertical
Displaying the SliderCell	– drawSelf:inView: – drawInside:inView: – drawBarInside:flipped: – drawKnob – drawKnob:
Modifying behavior	– setEnabled: – setContinuous: – isContinuous – setAltIncrementValue: – altIncrementValue

Tracking the mouse	+ prefersTrackingUntilMouseUp – trackMouse:inRect:ofView: – startTrackingAt:inView: – continueTracking:at:inView: – stopTracking:at:inView:mouseIsUp:
Archiving	– read: – write: – awake

Class Methods

prefersTrackingUntilMouseUp

+ (BOOL)**prefersTrackingUntilMouseUp**

Returns YES so a SliderCell can 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) 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 while tracking.

Instance Methods

altIncrementValue

– (double)**altIncrementValue**

Returns the amount that the SliderCell will alter its value when the user drags the knob one pixel with the Alternate key held down. If the Alternate-dragging feature isn't enabled, this method returns –1.0.

See also: – **setAltIncrementValue:**

awake

– **awake**

Retrieves the system images used to draw SliderCell knobs, and returns **self**. This message is sent from the **read:** method; you never send it yourself.

See also: – **read:**

calcCellSize:inRect:

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

Returns **self**, and by reference in *theSize* the minimum width and height needed to draw the SliderCell in *aRect*. If *aRect* too small to fit the knob and bezel, the width and height of *theSize* are set to 0.0.

If the SliderCell hasn't had its tracking rectangle set, this method will determine from *aRect* whether the SliderCell should be vertical or horizontal, and will set a vertical SliderCell's height to *aRect->size.height*, a horizontal SliderCell's width to *aRect->size.width*, and the other dimension of either type to the minimum SliderCell breadth.

If you draw your own knob on the SliderCell and that knob is not the same size as a standard SliderCell knob, or if you draw the SliderCell itself differently, you should override this method to take your knob's dimensions into account. You must also override **getKnobRect:flipped:** and **drawKnob:**.

Note: It's usually wrong to invoke the inherited **calcCellSize:** method. Instead, **calcCellSize:inRect:** should be used with a valid rectangle for displaying the SliderCell.

See also: – **getKnobRect:flipped:**, – **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 knob at the new position based on *currentPoint*.

Override this method if you want to change the way positioning is done.

See also: – **trackMouse:inRect:ofView:**, – **startTrackingAt:inView:**, – **stopTracking:at:inView:mouseIsUp:**

doubleValue

– (double)**doubleValue**

Returns the value of the SliderCell as a double-precision floating point number.

See also: – **setDoubleValue:**, – **floatValue**, – **intValue**, – **stringValue**

drawBarInside:flipped:

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

Draws the SliderCell's background bar, but not the bezel around it or the knob. *flipped* indicates whether the View's coordinate system is flipped or not. Returns **self**.

Override this method if you want to draw your own slider bar. Note, however, that the **setImage:** method allows you to conveniently customize the appearance of the SliderCell's background.

See also: – **drawInside:inView:**, – **drawSelf:inView:**, – **isFlipped** (View), – **setImage:**, – **lockFocus** (View)

drawInside:inView:

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

Draws the SliderCell's background bar and knob, along with the background title, but not the bezel. The PostScript focus must be locked on *controlView* when this message is sent. Returns **self**.

See also: – **drawBarInside:flipped:**, – **drawKnob**, – **drawSelf:inView:**, – **lockFocus** (View)

drawKnob

– **drawKnob**

Calculates the rectangle in which the knob should be drawn and invokes **drawKnob:** to actually draw the knob. The PostScript focus must be locked on the SliderCell's View when this message is sent. You never override this method; override **drawKnob:** instead.

See also: – **drawKnob:**, – **lockFocus** (View)

drawKnob:

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

Draws the knob in *knobRect*. The PostScript focus must be locked on the SliderCell's View when this message is sent.

Override this method and **getKnobRect:flipped:** if you want to draw your own knob. You should also override **calcCellSize:inRect:** if your knob is of a different size from the standard SliderCell knob.

See also: – **drawKnob**, – **getKnobRect:flipped:**, – **calcCellSize:inRect:**, – **isVertical**, – **lockFocus** (View)

drawSelf:inView:

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

Draws the SliderCell background bar (including the bezel) and knob. The knob is drawn at a position which reflects the current value of the SliderCell. This method doesn't invoke **drawInside:inView:**. The PostScript focus must be locked on *controlView* when this message is sent. Returns **self**.

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

If, for example, you want a SliderCell that 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.

See also: – **drawBarInside:flipped:**, – **drawKnob**, – **lockFocus** (View)

floatValue

– (float)**floatValue**

Returns the value of the SliderCell as a single-precision floating point number.

See also: – **setFloatValue:**, – **doubleValue**, – **intValue**, – **stringValue**

getKnobRect:flipped:

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

Returns **self**, and by reference in *knobRect* the rectangle into which the knob will be drawn. This rectangle is determined from the SliderCell's value in relation to its tracking rectangle and its minimum and maximum values. *flipped* indicates whether the SliderCell's View has a flipped coordinate system.

Override this method and **drawKnob:** if you want to draw your own knob. You should also override **calcCellSize:inRect:** if your knob is of a different size from the standard SliderCell knob (and be careful of setting the knob's width). Remember to take into account the flipping of the View in vertical SliderCells; otherwise, your knob might appear the correct distance from the wrong end.

See also: – **drawKnob:**, – **calcCellSize:inRect:**, – **isVertical**, – **isFlipped** (View)

image

– **image**

Returns the NXImage that the SliderCell displays as its background.

See also: – **setImage:**

init

– **init**

Initializes and returns the receiver, a new instance of SliderCell. Its value is set to 0.0, minimum value to 0.0, and maximum value to 1.0. New SliderCells are continuous by default.

This method is the designated initializer for SliderCell; override it if you create a subclass of SliderCell that performs its own initialization. You shouldn't use Cell's designated initializers, **initIconCell:** or **initTextCell:**, to initialize a SliderCell.

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

intValue

– (int)**intValue**

Returns the value of the SliderCell as an integer.

See also: – **setIntValue:**, – **doubleValue**, – **floatValue**, – **stringValue**

isContinuous

– (BOOL)**isContinuous**

Returns YES if the action is sent to the target continuously as mouse-dragged events occur while tracking, or on a mouse-up event; NO if the action is sent only on a mouse-up event.

See also: – **setContinuous:**

isOpaque

– (BOOL)**isOpaque**

Returns YES, since a SliderCell always draw over every pixel in its frame.

See also: – **isOpaque** (Cell)

isVertical

– (int)**isVertical**

Returns 1 if the SliderCell is vertical, 0 if it's horizontal, and –1 if the orientation can't be determined (because the SliderCell hasn't been drawn in a View, for example). A SliderCell is vertical if its height is greater than its width.

knobThickness

– (NXCoord)**knobThickness**

Returns the thickness of the SliderCell's knob (that is, its extent along the bar's length) in the SliderCell's coordinate system.

See also: – **setKnobThickness:**

maxValue

– (double)**maxValue**

Returns the maximum value of the SliderCell.

See also: – **setMaxValue:**, – **minValue**

minValue

– (double)**minValue**

Returns the minimum value of the SliderCell.

See also: – **setMinValue:**, – **maxValue**

read:

– **read:**(NXTypedStream *)*stream*

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

See also: – **write:**, – **awake**

setAltIncrementValue:

– **setAltIncrementValue:**(double)*incValue*

Sets the amount that the SliderCell will alter its value when the user drags the knob one pixel with the Alternate key held down. *incValue* should be greater than 0.0, and less than the SliderCell's maximum value; it can also be –1, in which case this feature is disabled. Normally, you'll want to use this method with *incValue* less than 1.0, so the knob will move more slowly than the mouse.

See also: – **altIncrementValue**, – **maxValue**

setContinuous:

– **setContinuous:**(BOOL)*flag*

If *flag* is YES, the SliderCell will send its action to its target continuously as mouse-dragged events occur while tracking, or on a mouse-up event. If NO, the SliderCell will send its action only on a mouse-up event. The default is YES. Returns **self**.

See also: – **isContinuous**

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**.

See also: – **doubleValue**, – **setFloatValue:**, – **setIntValue:**, – **setStringValue:**

setEnabled:

– **setEnabled:**(BOOL)*flag*

If flag is YES, the SliderCell will become enabled; if NO, the SliderCell will become disabled. A disabled SliderCell draws its non-image background in light gray. An enabled SliderCell draws its non-image background in 50% gray.

See also: – **isEnabled** (ActionCell)

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**.

See also: – **floatValue**, – **setDoubleValue:**, – **setIntValue:**, – **setStringValue:**

setImage:

– **setImage:***image*

Sets the NXImage used as the SliderCell's background. This method doesn't scale the NXImage. Returns **self**.

See also: – **image**

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**.

See also: – **intValue**, – **setDoubleValue:**, – **setFloatValue:**, – **setStringValue:**

setKnobThickness:

– **setKnobThickness:**(NXCoord)*aFloat*

Sets the thickness of the SliderCell's knob (that is, its extent along the bar's length) in its own coordinate system. *aFloat* must be greater than 0.0, and shouldn't be greater than the Slider's length. If the knob thickness changes, the SliderCell's inside is redrawn. Returns **self**.

See also: – **knobThickness**

setMaxValue:

– **setMaxValue:**(double)*aDouble*

Sets the maximum value of the SliderCell to *aDouble* and returns **self**. If the maximum value changes, the SliderCell's inside is redrawn to reposition the knob relative to the new maximum.

See also: – **maxValue**, – **setMinValue:**

setMinValue:

– **setMinValue:**(double)*aDouble*

Sets the minimum value of the SliderCell to *aDouble* and returns **self**. If the minimum value changes, the SliderCell's inside is redrawn to reposition the knob relative to the new minimum.

See also: – **minValue**, – **setMaxValue:**

setStringValue:

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

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

Note: SliderCell doesn't override the **setStringValueNoCopy:** or **setStringValueNoCopy:shouldFree:** methods; you shouldn't use those methods with a SliderCell.

See also: – **stringValue**, – **setDoubleValue:**, – **setFloatValue:**, – **setIntValue:**

setTitle:

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

Sets the title drawn over the SliderCell's background to *aString*. Returns **self**.

See also: – **setTitleNoCopy:**, – **title**

setTitleCell:

– **setTitleCell:***aCell*

Sets the Cell used to draw the SliderCell's background title. *aCell* should be an instance of TextFieldCell (or of a subclass). Doesn't redraw the SliderCell; further, a **setTitle:** message is required to display a title, even if aCell already has a string value. Returns the old Cell.

See also: – **titleCell**, – **setTitle:**

setTitleColor:

– **setTitleColor:**(NXColor)*color*

Sets the color used to draw the SliderCell's background title, redraws the SliderCell's inside, and returns **self**. The default is to draw in a gray level of 0.0 (NX_BLACK).

See also: – **titleColor**, – **setTitleGray:**

setTitleFont:

– **setTitleFont:***fontObject*

Sets the Font used to draw the SliderCell's background title and redraws the SliderCell's inside. The default font is the default system font as set by the user (with the Preferences application), and its size is 12.0 point. Returns **self**.

See also: – **titleFont**

setTitleGray:

– **setTitleGray:**(float)*aFloat*

Sets the gray value used to draw the SliderCell's background title, redraws the SliderCell's inside, and returns **self**. The default gray level is 0.0 (NX_BLACK).

See also: – **titleGray**, – **setTitleColor:**

setTitleNoCopy:

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

Sets the title drawn over the SliderCell's background to *aString*, but doesn't copy the string. Returns **self**.

See also: – **setTitle:**, – **title**

startTrackingAt:inView:

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

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

See also: – **trackMouse:inRect:ofView:**, – **continueTracking:at:inView:**, – **stopTracking:at:inView:mouseIsUp:**

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**.

See also: – **trackMouse:inRect:ofView:**, – **startTrackingAt:inView:**, – **continueTracking:at:inView:**

stringValue

– (const char *)**stringValue**

Returns a string representing the value of the SliderCell. The floating point format is applied when generating the string representation.

See also: – **setStringValue:**, – **doubleValue**, – **floatValue**, – **intValue**, – **setFloatingPointFormat:left:right:** (Cell)

title

– (const char *)**title**

Returns the string used as the SliderCell's background title. The title is drawn over the SliderCell's background. Returns **self**.

See also: – **setTitle:**

titleCell

– **titleCell**

Returns the TextFieldCell used to draw the SliderCell. If the SliderCell doesn't have a title, a new TextFieldCell is created and returned. This doesn't result in a title getting set.

See also: – **setTitleCell:**

titleColor

– (NXColor)**titleColor**

Returns the color used to draw the SliderCell's background title. The default is to draw in a gray level of 0.0 (NX_BLACK). Returns **self**.

See also: – **setTitleColor:**, – **titleGray**

titleFont

– **titleFont**

Returns the Font used to draw the SliderCell's title. The default font is the default system font as set by the user (with the Preferences application), and its size is 12.0 point.

See also: – **setTitleFont:**

titleGray

– (float)**titleGray**

Returns the gray value used to draw the SliderCell's background title. The default gray level is 0.0 (NX_BLACK). Returns **self**.

See also: – **setTitleGray:**, – **titleColor**

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. Since SliderCell responds YES to **prefersTrackingUntilMouseUp**, this method will be invoked with a NULL *cellFrame*. Returns YES if the mouse goes up, NO otherwise.

If the SliderCell is continuous, then the action will be continuously sent to the target as the mouse is dragged. If *cellFrame* isn't the same *cellFrame* that was passed to the last **drawSelf:inView:**, then this method doesn't track.

See also: **– startTrackingAt:inView:**, **– continueTracking:at:inView:**, **– stopTracking:at:inView:mouseIsUp:**, **– setContinuous:**

write:

– **write:**(NXTypedStream *)*stream*

Writes the receiving SliderCell to the typed stream *stream*. Returns **self**.

See also: **– read:**

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 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 **mach/message.h**.
- 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

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:
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:

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:**

free

– **free**

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

init

– **init**

Initializes a newly allocated Speaker instance. 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**.

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:**

read:

– **read:**(NXTypedStream *)*stream*

Reads the Speaker from the typed stream *stream*. The Speaker's send-port and reply-port will both be PORT_NULL. Returns **self**.

See also: – **write:**

replyPort

– (port_t)**replyPort**

Returns the port where the Speaker expects to receive return messages. 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, an **openFile:ok:** message could be sent as follows:

```
int      msgDelivered, wasOK;

msgDelivered = [mySpeaker selectorRPC:"openFile:ok:"
                          paramTypes:"cI","/usr/foo",
                          &wasOK]
```

params is a character string, "cI" 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, –1 if it was delivered but wasn't understood or couldn't be delegated, and NX_INCORRECTMESSAGE if the RPC succeeds but the selector is not implemented at the other end.

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.

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**. However, 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**

write:

– **write:**(NXTypedStream *)*stream*

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

See also: – **read:**

Text

Inherits From: View : Responder : Object

Conforms To: NXChangeSpelling
NXIgnoreMisspelledWords
NXReadOnlyTextStream
NXSelectText

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. For convenience, such a Text object will be called a *rich Text object*.

A rich Text object can use RTF (Rich Text Format) as an interchange format. Not all RTF control words are supported: On input, a Text object ignores any control word it doesn't recognize; some of those it can read and interpret it doesn't write out. These are the RTF control words that a Text object recognizes.

Control Word	Read	Write
\ansi	yes	(see note below)
\b	yes	yes
\cb	yes	yes
\cf	yes	yes
\colortbl	yes	yes
\dnn	yes	yes
\fin	yes	yes
\fn	yes	yes
\fonttbl	yes	yes
\fsn	yes	yes
\i	yes	yes
\lin	yes	yes
\margrn	yes	yes
\paperwn	yes	yes
\mac	yes	no
\margln	yes	yes
\par	yes	yes
\pard	yes	no
\pca	yes	no
\qc	yes	yes
\ql	yes	yes
\qr	yes	yes
\sn	yes	no
\tab	yes	yes
\upn	yes	yes

Note: A Text object writes 8-bit characters in the NeXTSTEP encoding, which differs somewhat from the ANSI character set. See the appendix "Keyboard Event Information" for more details

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

Graphics are embedded in the text in either of two ways: programmatically or directly through user actions. The programmatic approach is discussed first.

In the programmatic approach, you add an object—generally a subclass of Cell—to the text. This object will manage the graphic image by drawing it when appropriate. Although Cell subclasses are commonly used, the only requirement is that the embedded object

responds 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:

You place the object in the text by sending the Text object a **replaceSelWithCell:** message.

A Text object displays a graphic in its text by sending the managing object a **drawSelf:inView:** message. To record the graphic to a file or to the pasteboard, the Text object sends the managing object a **writeRichText:forView:** message. The 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 the 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.

An alternate means of adding a image to the text is for the user to drag an EPS or TIFF file icon directly into a Text object. The Text object automatically creates an object to manage the display of the image. This feature requires a rich Text object that has been configured to receive dragged images (see **setGraphicsImportEnabled:**).

Images that have been imported in this way can be written as RTFD documents, the file format that Edit uses for text that contains images. RTFD documents use a file package, or directory, to store the components of the document (the "D" stands for "directory"). (See **saveRTFDTo:removeBackup:errorHandler:** and **openRTFDFrom:**.) The file package has the name of the document plus a ".rtfd" extension. It always contains a file called **TXT.rtf** for the text of the document, and one or more TIFF or EPS files for the images. A Text object can serialize the information in an RTFD document to a stream (see **writeRTFDTo:**) and deserialize it from a stream (see **readRTFDFrom:**).

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

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 for the vertical bar that marks the insertion point.
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.

Adopted Protocols

NXChangeSpelling	– changeSpelling:
NXIgnoreMisspelledWords	– spellDocumentTag
NXReadOnlyTextStream	– openTextStream – seekToCharacterAt:relativeTo: – readCharacters:count: – currentCharacterOffset – isAtEOTS – closeTextStream
NXSelectText	– selectCharactersFrom:to: – selectionCharacterCount – readCharactersFromSelection:count: – makeSelectionVisible

Method Types

Initializing the class object	+ setDefaultFont: + getDefaultFont + excludeFromServicesMenu: + registerDirective:forClass: + initialize
Initializing a new Text object	– initFrame: – initFrame:text:alignment:
Freeing a 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: – offsetFromPosition: – positionFromOffset:

Setting, reading, and writing the text

- setText:
- readText:
- startReadingRichText
- readRichText:
- readRichText:atPosition:
- readRTFDFrom:
- finishReadingRichText
- openRTFDFrom:
- saveRTFDTo:removeBackup:errorHandler:
- writeText:
- writeRichText:
- writeRichText:from:to:
- writeRTFDSelectionTo:
- writeRTFDTo:
- stream
- firstTextBlock
- getParagraph:start:end:rect:
- getSubstring:start:length:
- byteLength
- charLength
- textLength

Setting editability

- setEditable:
- isEditable

Allowing multiple fonts and paragraph styles

- setMonoFont:
- isMonoFont

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: – replaceSelWithRTFD: – scrollSelToVisible
Setting the font	– setFontPanelEnabled: – isFontPanelEnabled – changeFont: – setFont: – font – setFont:paraStyle: – setSelFont: – setSelFontFamily: – setSelFontSize: – setSelFontStyle: – setSelFont:paraStyle:
Checking spelling	– checkSpelling: – showGuessPanel:
Managing the ruler	– toggleRuler: – isRulerVisible
Finding text	– findText:ignoreCase:backwards:wrap:

Modifying graphic attributes	– setBackgroundGray: – backgroundGray – setBackgroundColor: – backgroundColor – setSelGray: – selGray – runGray: – setSelColor: – selColor – runColor: – 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:
Handling graphics within the text	+ registerDirective:forClass: – replaceSelWithCell: – setLocation:ofCell: – getLocation:ofCell: – setGraphicsImportEnabled: – isGraphicsImportEnabled

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 (thus, this method would return the same value as **charLength** or **textLength**). Subclasses of Text that use more than one byte per character should override this method to return the number of bytes used to store the text.

See also: – **charLength**, – **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:
– read:
– renewFont:text:frame:tag:
– renewRuns:text:frame:tag:
– setFont:paraStyle:
– readText:
– renewFont:size:style:text:frame:tag:
– setFont:
– setParaStyle:
– 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:**

charLength

– (int)**charLength**

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. The **charLength** and **textLength** methods are identical; the related method **byteLength** returns the length of the text in bytes, which, depending on the number of bytes used to store a character, may return a larger value.

See also: – **byteLength**, – **textLength**, – **getSubstring:start:length:**

charWrap

– (BOOL)**charWrap**

Returns 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 the 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)

findText:ignoreCase:backwards:wrap:

– (BOOL)**findText:**(const char *)*string*
ignoreCase:(BOOL)*ignoreCaseflag*
backwards:(BOOL)*backwardsflag*
wrap:(BOOL)*wrapflag*

Searches for *string* in the text, starting at the insertion point. If *ignoreCaseflag* is YES, the search is case-insensitive. If *backwardsflag* is NO, the search proceeds forward through the text. If *wrapflag* is YES, upon reaching the end of the text, the search loops back to the start. If the string is found, it's highlighted and—if the Text object is the document view of a ScrollView—the selection is scrolled into view. Returns YES, if *string* is found, NO otherwise.

This method searches for the literal string; regular expression substitutions and wildcard characters aren't supported.

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:**, – **calcCellSize:** (Cell)

getLocation:ofView:

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. The default value is YES.

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:**

isGraphicsImportEnabled

– (BOOL)**isGraphicsImportEnabled**

Returns YES if the Text object will import TIFF and EPS images dragged into it by the user. The default value is NO.

See also: – **setGraphicsImportEnabled:**

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 permits only one font and paragraph style for its text, NO if not. The default value is YES.

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. The default value is YES.

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)

offsetFromPosition:

– (int)**offsetFromPosition:**(int)*charPosition*

Returns the byte offset corresponding to the character position *charPosition* in the Text object's text. In the standard software release, where each character is represented by a byte, a character's position and its byte offset are identical.

See also: – **positionFromOffset:**, – **positionFromLine:**, – **lineFromPosition:**

openRTFDFrom:

– (NXRTFDError)**openRTFDFrom:**(const char *)*path*

Opens the RTFD file package specified by *path*. The last element in the path must be the name of the RTFD directory—for example, "/tmp/MyFile.rtfd"—not the name of the RTF document within the directory. On success, the Text object's contents are replaced with the text and images found in the file package, and the new contents are displayed.

See also: – **readRTFDFrom:**, – **replaceSelWithRTFD:**, – **writeRTFDSelectionTo:**, – **writeRTFDTo:**

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**

positionFromOffset:

– (int)**positionFromOffset:**(int)*offset*

Returns the character position corresponding to a byte offset into the Text object's text. The character position is determined by counting characters from the beginning of the Text object, with the first character designated as 0. In the standard software release, where each character is represented by a byte, a character's position and its byte offset are identical.

See also: – **offsetFromPosition:**, – **positionFromLine:**, – **lineFromPosition:**

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. 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. Returns **self**.

readRTFDFrom:

– **readRTFDFrom:**(NXStream *)*stream*

Reads the RTFD data contained in *stream*. The Text object's contents are replaced with the text and images found in the stream, and the new contents are displayed. Returns **self** if the data is successfully read from the stream; otherwise, returns **nil**.

See also: – **openRTFDFrom:**, – **replaceSelWithRTFD:**, – **writeRTFDSelectionTo:**, – **writeRTFDTo:**

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*, assuming the Text object accepts rich text (see **setMonoFont:**). 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:**, – **replaceSelWithRTFD:**

replaceSelWithRTFD:

– **replaceSelWithRTFD:**(NXStream *)*stream*

Replaces the current selection with RTFD data from *stream*, assuming the Text object accepts rich text (see **setMonoFont:**). A **replaceSelWithRTFD:** message is sent in response to pasting RTFD data from the pasteboard.

After replacing the selection, this method rewraps and redisplays the text. On error **replaceSelWithRTFD:** returns **nil**; otherwise, returns **self**.

See also: – **replaceSel:**, – **replaceSel:length:runs:**, – **replaceSelWithRichText:**

replaceSelWithView:

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**.

runColor:

– (NXColor)**runColor:**(NXRun *)*run*

Returns the color of the specified text run. By definition, a run can have no more than one color.

See also: – **runGray:**

runGray:

– (float)**runGray:**(NXRun *)*run*

Returns the gray value for the specified text run. By definition, a run can have no more than one gray value.

See also: – **runColor:**

saveRTFDTo:removeBackup:errorHandler:

– (NXRTFDError)**saveRTFDTo:**(const char *)*path*
removeBackup:(BOOL)*flag*
errorHandler:*handler*

Saves the contents (text and images) of the Text object to the file package specified by *path*, for example, **/text/Document.rtfd**. The text and images are saved in separate files within the file package, with the text file being named **TXT.rtf**.

The save operation proceeds in several steps: First, the document in memory is saved to a temporary file package (using the above example, **/text/Document.rtfd#**). Second, the earlier version of the document—if one exists—is renamed as the backup (**/text/Document.rtfd** becomes **/text/Document.rtfd~**). Then, the temporary file package is renamed (**/text/Document.rtfd#** becomes **/text/Document.rtfd**). Finally, if *flag* is YES, the backup is deleted.

handler is an error handling object that implements the **attemptOverwrite:** method (see the NXRTFDErrorHandler protocol specification for details). If the user doesn't have search permission for a component of *path*, an **attemptOverwrite:** message is sent to the error handler. If the error handler returns YES, the Text object will attempt to write the file; otherwise, the save operation is aborted.

This method returns a value indicating its success or the reason for its failure:

Return Values

NX_RTFDErrorNone
NX_RTFDErrorSaveAborted,
NX_RTFDErrorUnableToWriteFile,
NX_RTFDErrorUnableToCloseFile,
NX_RTFDErrorUnableToCreatePackage,
NX_RTFDErrorUnableToCreateBackup,
NX_RTFDErrorUnableToDeleteBackup,
NX_RTFDErrorUnableToDeleteTemp,
NX_RTFDErrorUnableToDeleteOriginal,
NX_RTFDErrorFileDoesntExist,
NX_RTFDErrorUnableToReadFile,
NX_RTFDErrorInsufficientAccess,
NX_RTFDErrorMalformedRTFD

See also: – openRTFDFrom:, – readRTFDFrom:, – writeRTFDTo:

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. This method works by invoking the **scrollRectToVisible:** method, which Text inherits from View. Returns **self**.

selColor

– (NXColor)**selColor**

Returns the color of the selected text.

See also: – **setSelColor:**, – **setSelGray:**, – **setBackgroundGray:**, – **backgroundGray**, – **setTextGray:**, – **textGray**

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. This method works by invoking the **selectAll:** method. Returns **self**.

See also: – **selectAll:**, – **setSel::**

selGray

– (float)**selGray**

Returns the gray value of the selected text.

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**

setGraphicsImportEnabled:

– **setGraphicsImportEnabled:**(BOOL)*flag*

Sets whether the Text object will import TIFF and EPS images dragged into it by the user. By default, Text objects refuse to import such images.

See also: – **isGraphicsImportEnabled**

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:**

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**

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.

The **textLength** and **charLength** methods are identical; the related method **byteLength** returns the length of the text in bytes, which, depending on the number of bytes used to store a character, may return a larger value.

See also: – **byteLength**, – **charLength**, – **getSubstring:start:length:**

toggleRuler:

– **toggleRuler:***sender*

Controls the display of the ruler. This method has effect only if the receiving Text object is a rich Text object (see **setMonoFont:**) and 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:**, – **setMonoFont:**

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. Returns **self**.

See also: – **writeText:**, – **readText:**

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:**

writeRTFDSelectionTo:

– **writeRTFDSelectionTo:**(NXStream *)*stream*

Writes the Text object's selection to *stream*. If the selection consists of text and images from files in an RTFD directory, this data is serialized into the stream. Returns **self**.

See also: – **openRTFDFrom:**, – **readRTFDFrom:**, – **replaceSelWithRTFD:**, – **writeRTFDTo:**

writeRTFDTo:

– **writeRTFDTo:**(NXStream *)*stream*

Writes the Text object's contents to *stream*. If the Text object is storing RTFD data (that is, text and images) this data is serialized into the stream. The counterpart method, **readRTFDFrom:**, can restore the Text object's contents from such serialized data. Returns **self**.

See also: – **openRTFDFrom:**, – **readRTFDFrom:**, – **replaceSelWithRTFD:**, – **writeRTFDSelectionTo:**

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 (for example, Tab or Return) that caused the Text object to cease being the first responder and is represented by constants such as NX_TAB and NX_RETURN. (See "Types and Constants" for a complete list of these constants.) 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 control words.

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 YES. 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 YES. 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.

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, the Text 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:**

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:**

TextField

Inherits From: Control : View : Responder : Object

Declared In: appkit/TextField.h

Class Description

A TextField is a Control object that can display a piece of text that a user can select or edit, and which sends an action message to its target if the user hits the Return key while editing. A TextField can also be linked to other TextFields, so that when the user presses Tab or Shift-Tab, the object assigned as the "next" or "previous" field gets a message to select its text.

A TextField is a good alternative to a Text object for small regions of editable text, since the display of the TextField is achieved by using a global Text object shared by objects all over your application, which saves on memory usage. Each Window also has a Text object used for editing of TextFields (and TextFieldCells in Matrices). A Window's global Text object is called a *field editor*, since it's attached as needed to a TextField to perform its editing. TextField allows you to specify an object to act as an indirect delegate to the field editor; the TextField itself acts as the Text delegate if it needs to, then passes the delegate method on to its own Text delegate.

Instance Variables

id **nextText**;
id **previousText**;
id **textDelegate**;
SEL **errorAction**;

nextText	The object whose text is selected when Tab is pressed.
previousText	The object whose text is selected when Shift-Tab is pressed.

textDelegate	Delegate for Text object delegate methods.
errorAction	Message sent to the 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:
Setting user access to text	– setSelectable: – isSelectable – setEditable: – isEditable
Editing Text	– selectText:
Setting Tab key behavior	– setNextText: – nextText – setPreviousText: – previousText
Assigning a Text delegate	– setTextDelegate: – textDelegate
Text object delegate methods	– textWillChange: – textDidChange: – textDidGetKeys:isEmpty: – textWillEnd: – textDidEnd:endChar:
Setting the TextField's value	– setFloatValue: (Control) – floatValue (Control) – setDoubleValue: (Control) – doubleValue (Control) – setIntValue: (Control) – intValue (Control) – setStringValue: (Control) – setStringValueNoCopy: (Control) – setStringValueNoCopy:shouldFree: (Control) – stringValue (Control)

Modifying graphic attributes	– setTextColor: – textColor – setTextGray: – textGray – setBackgroundColor: – backgroundColor – setBackgroundGray: – backgroundGray – setBackgroundTransparent: – isBackgroundTransparent – setBezeled: – isBezeled – setBordered: – isBordered
Target and action	– setErrorAction: – errorAction
Resizing a TextField	– sizeTo::
Handling events	– acceptsFirstResponder – mouseDown:
Archiving	– read: – write:

Class Methods

setCellClass:

+ **setCellClass:***classId*

Configures the TextField class to use instances of *classId* for its Cells. *classId* should be the **id** of a subclass of TextFieldCell, obtained by sending the **class** message to either the Cell subclass object or to an instance of that subclass. The default Cell class is TextFieldCell. Returns **self**.

For more on how to safely set a Cell class for your subclass of TextField, see "Creating New Controls" in the Control class specification.

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 color used to draw the background.

See also: – **setBackgroundColor:**, – **backgroundGray**

backgroundGray

– (float)**backgroundGray**

Returns the gray level used to draw the background. If the gray level is less than 0, then the background is transparent.

See also: – **setBackgroundGray:**, – **backgroundColor**

errorAction

– (SEL)**errorAction**

Returns the action sent to the target of the TextField when the user enters an illegal value for the Cell type (as set by Cell's **setEntryType:** method and checked by Cell's **isEntryAcceptable:** method).

See also: – **setErrorAction:**, – **setEntryType:** (Cell), – **isEntryAccectable:** (Cell)

initFrame:

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

Initializes and returns the receiver, a new instance of TextField, with default parameters in the given frame. The string value is set to be empty (not NULL), and the TextField has no target, action, error action, or Text delegate. The TextField is editable, drawn with a bezel,

a white background, and black text; the alignment is NX_LEFTALIGNED. The font is set to the user's system font, and the font size is 12.0 point. This is the designated initializer for the TextField class.

isBackgroundTransparent

– (BOOL)**isBackgroundTransparent**

Returns YES if the background of the TextField is transparent (that is, if the background gray is less than 0).

See also: – **setBackgroundTransparent:**

isBezeled

– (BOOL)**isBezeled**

Returns YES if the text is drawn in a bezeled frame.

See also: – **setBezeled:**, – **isBordered**

isBordered

– (BOOL)**isBordered**

Returns YES if the text has a solid black border around it.

See also: – **setBordered:**, – **isBezeled**

isEditable

– (BOOL)**isEditable**

Returns YES if the text is editable and selectable, NO if the text is not editable (though it may be selectable).

See also: – **setEditable:**, – **isSelectable**

isSelectable

– (BOOL)**isSelectable**

Returns YES if the text is selectable, NO otherwise. Selectable text isn't necessarily editable.

See also: – **setSelectable:**, – **isEditable**

mouseDown:

– **mouseDown:**(NXEvent *)*theEvent*

Overrides the Control method to begin editing or select text if the TextField allows it. You never invoke this method directly, but may override it to implement subclasses of the TextField class. Returns **self**.

See also: – **isEditable**, – **isSelectable**

nextText

– **nextText**

Returns the object whose text is selected when the user presses Tab while editing the TextField. If that object responds to the **selectText:** message, the current TextField is deactivated and **selectText:** is send to the next text.

See also: – **setNextText:**, – **previousText**

previousText

– **previousText**

Returns the object that is selected when the user presses Shift-Tab while editing the TextField. If that object responds to the **selectText:** message, the current TextField is deactivated and **selectText:** is sent to the previous text.

See also: – **setPreviousText:**, – **nextText**

read:

– **read:**(NXTypedStream *)*stream*

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

See also: – **write:**

selectText:

– **selectText:***sender*

Selects the entire contents of the receiving TextField if it is editable or selectable. If the TextField isn't in a View hierarchy, it has no effect. Returns **self**.

See also: – **isEditable**, – **isSelectable**

setBackgroundColor:

– **setBackgroundColor:**(NXColor)*aColor*

Sets the background color for the TextField. Returns **self**.

See also: – **backgroundColor**, – **setBackgroundGray:**

setBackgroundGray:

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

Sets the gray level that will be used to draw the background. If *value* is less than 0.0, no background will be drawn. If the TextField is editable, it should have a background gray greater than or equal to 0.0. Returns **self**.

See also: – **backgroundGray**, – **setBackgroundColor:**

setBackgroundTransparent:

– **setBackgroundTransparent:**(BOOL)*flag*

If *flag* is YES, sets the background gray of the TextField to transparent (a negative value); if NO, sets the background gray to NX_WHITE. Returns **self**.

See also: – **setBackgroundGray:**

setBezeled:

– **setBezeled:**(BOOL)*flag*

If flag is YES, the TextFieldCell is drawn with a bezel around the edge; if NO, nothing is drawn around the text. Bezels and borders are mutually exclusive. If the current background gray is transparent, it's changed to NX_WHITE. Bezeled transparent TextFields look rather strange, but if you want to have one, invoke **setBackgroundGray:** with –1.0 *after* invoking **setBezeled:**.

See also: – **isBezeled**, – **setBordered:**, – **setBackgroundGray:**

setBordered:

– **setBordered:**(BOOL)*flag*

If *flag* is YES, a 1-pixel black border will be drawn around the text; if NO, nothing is drawn around the text. Borders and bezels are mutually exclusive. Does not affect the background gray level or color. Returns **self**.

See also: – **isBordered**, – **setBezeled:**

setEditable:

– **setEditable:**(BOOL)*flag*

If *flag* is YES, then the text in the TextField is made both editable and selectable. If NO, the text can't be edited, and is restored to its previous selectable state. For example, if a TextField is set selectable but not editable, then made editable for a time, then made not editable again, it will still be selectable. To guarantee that text will be neither editable nor selectable, simply turn off selectability explicitly. Returns **self**.

See also: – **isEditable**, – **setSelectable:**

setEnabled:

– **setEnabled:**(BOOL)*flag*

Makes the TextField enabled (able to accept mouse clicks and keystrokes) according to *flag*. Redraws the text of the cell if autodisplay is on and the enabled state changes. Returns **self**.

See also: – **isEnabled** (Control)

setErrorAction:

– **setErrorAction:**(SEL)*aSelector*

Sets the action sent to the target of the TextField when the user enters an illegal value for the Cell's entry type (as set by Cell's **setEntryType:** method and checked by Cell's **isEntryAcceptable:** method). Returns **self**.

See also: – **errorAction**, – **setEntryType:** (Cell), – **isEntryAccectable:** (Cell)

setNextText:

– **setNextText:***anObject*

Sets up *anObject* as the object whose text will be selected when the user presses Tab while editing the TextField's text. *anObject* should respond to the **selectText:** message. If *anObject* also responds to both **selectText:** and **setPrevious:**, it's sent **setPreviousText:** with the receiving TextField as the argument; this builds a two-way connection, so that pressing Tab in the TextField selects *anObject*'s text, and pressing Shift-Tab in *anObject* selects the TextField's text. Returns **self**.

See also: – **nextText**, – **setPreviousText:**, – **selectText:**

setPreviousText:

– **setPreviousText:***anObject*

Sets up *anObject* as the object whose text will be selected when the user presses Shift-Tab while editing the TextField's text. *anObject* should respond to the **selectText:** message. Your code shouldn't need to use this method directly, since it's invoked automatically by **setNextText:**. In deference to **setNextText:**, this method doesn't build a two-way connection. Returns **self**.

See also: – **previousText**, – **setNextText:**, – **selectText:**

setSelectable:

– **setSelectable:**(BOOL)*flag*

If *flag* is YES, then the TextField is made selectable but *not* editable (use **setEditable:** to make text both selectable and editable). If NO, then the text is made neither editable nor selectable. Returns **self**.

See also: – **isSelectable**, – **setEditable:**

setTextColor:

– **setTextColor:**(NXColor)*aColor*

Sets the color used to draw the text. Returns **self**.

See also: – **textColor**, – **setTextGray:**

setTextDelegate:

– **setTextDelegate:***anObject*

Sets the object to which the TextField will pass along any messages from the field editor. These messages include **text:isEmpty:**, **textWillEnd:**, **textDidEnd:endChar:**, **textWillChange:**, and **textDidChange:**. Returns **self**.

See also: – **textDelegate**, Text delegate methods

setTextGray:

– **setTextGray:**(float)*value*

Sets the gray level used to draw the text. Returns **self**.

See also: – **textGray**, – **setTextColor:**

sizeTo::

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

Resizes the TextField to *width* and *height*, aborting any editing in the TextField. After the TextField is resized, this method reselects all the text. Returns **self**.

textColor

– (NXColor)**textColor**

Returns the color used to draw the text. Returns **self**.

See also: – **setTextColor:**, – **textGray**

textDelegate

– **textDelegate**

Returns the object that receives messages passed on by the TextField from the field editor.

See also: – **setTextDelegate:**

textDidChange:

– **textDidChange:***textObject*

Passes this message on, with the same argument, to the TextField's Text delegate. Override this method if you want your subclass of TextField to act as the field editor's delegate. Returns **self**.

See also: – **textDidChange:** (Text delegate)

textDidEnd:endChar:

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

Invoked by *textObject* (the field editor) when text editing ends. Passes this message on, with the same argument, to the TextField's Text delegate, then ends editing for the field editor and checks *whyEnd* to see if an action key (Return or Tab) was pressed. If Return was pressed, the action message is sent to the target. If Tab was pressed, **selectText:** is sent to the next text if there is one and it responds, or to **self** if not. If Shift-Tab was pressed, **selectText:** is sent to the previous text if there is one and it responds, or to **self** if not. Returns the object sent the **selectText:** message.

You may want to override this method to interpret more characters (such as the Enter or Escape keys) in ending editing.

See also: – **sendAction:to:** (Control), – **setNextText:**, – **setPreviousText:**, – **textDidEnd:endChar:** (Text delegate)

textDidGetKeys:isEmpty:

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

Passes this message on, with the same argument, to the TextField's Text delegate. Override this method if you want your subclass of TextField to act as the field editor's delegate. Returns **self**.

See also: – **textDidGetKeys:isEmpty:** (Text delegate)

textGray

– (float)**textGray**

Returns the gray level used to draw the text. Returns **self**.

See also: – **setTextGray:**, – **textColor**

textWillChange:

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

Invoked automatically during editing to determine if it is okay to edit the TextField. This method checks whether the TextField is editable and sends **textWillChange:** to the TextField's Text delegate to allow it to respond. Returns YES if the text isn't editable, NO if the text is editable but the TextField's Text delegate doesn't respond to **textWillChange:**, and the text delegate's return value for **textWillChange:** if the TextField's Text delegate responds to it.

See also: – **setEditable:**, – **setTextDelegate:**, – **textWillChange:** (Text delegate)

textWillEnd:

– (BOOL)**textWillEnd:***textObject*

Invoked automatically before text editing ends. Checks the text by sending **isEntryAcceptable:** to the TextField's Cell. If the entry isn't acceptable, sends the error action to the target. This method is then passed on to the TextField's Text delegate with the same argument. The return value is based on whether the entry is acceptable and on the return value from the TextField's Text delegate. If the delegate responds to **textWillEnd:**, this method returns NO if the entry is acceptable and the delegate returns NO. Otherwise this method returns YES to indicate that editing shouldn't end, and generates a beep to indicate an error in the entry.

See also: – **isEntryAcceptable:** (Cell), – **setTextDelegate:**,
– **textWillEnd:** (Text delegate)

write:

– **write:**(NXTypedStream *)*stream*

Writes the receiving TextField to the typed stream *stream*. Returns **self**.

See also: – **read:**

TextFieldCell

Inherits From: ActionCell : Cell : Object

Declared In: appkit/TextFieldCell.h

Class Description

A TextFieldCell is simply a text Cell that keeps track of it background and text colors. 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 colors.

Instance Variables

float **backgroundGray**;
float **textGray**;

backgroundGray	The background gray level.
textGray	The text gray level.

Method Types

Initializing a new TextFieldCell	– init – initTextCell:
Copying a TextFieldCell	– copyFromZone:

Setting the TextFieldCell's value	– setFloatValue: (Cell) – floatValue (Cell) – setDoubleValue: (Cell) – doubleValue (Cell) – setIntValue: (Cell) – intValue (Cell) – setStringValue: (Cell) – setStringValueNoCopy: (Cell) – setStringValueNoCopy:shouldFree: (Cell) – stringValue (Cell)
Modifying Graphic Attributes	– setTextColor: – textColor – setTextGray: – textGray – setBackgroundColor: – backgroundColor – setBackgroundGray: – backgroundGray – setBackgroundTransparent: – isBackgroundTransparent – setTextAttributes: – setBezeled: – isOpaque
Displaying	– drawSelf:inView: – drawInside:inView:
Tracking the Mouse	– trackMouse:inRect:ofView:
Archiving	– read: – write:

Instance Methods

backgroundColor

– (NXColor)**backgroundColor**

Returns the color used to draw the background.

See also: **– setBackgroundColor:**, **– backgroundGray**

backgroundGray

– (float)**backgroundGray**

Returns the gray level used to draw the background. If the gray level is less than 0, then the background is transparent.

See also: – **setBackgroundGray:**, – **backgroundColor**

copyFromZone:

– **copyFromZone:**(NXZone *)*zone*

Creates and returns a new TextFieldCell as a copy of the receiver, allocated from *zone*.

drawInside:inView:

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

Draws the inside of the TextFieldCell (the background and text, but not the bezel or border). This method is invoked from **drawSelf:inView:** and also from Control and its subclasses' **drawCellInside:** method. If you subclass TextFieldCell, and you override **drawSelf:inView:**, then you should override this method as well. Returns **self**.

See also: – **drawSelf:inView:**

drawSelf:inView:

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

Draws the TextFieldCell's background, text, and border or bezel. Returns **self**.

See also: – **drawInside:inView:**

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.

See also: – **initTextCell:**

initTextCell:

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

Initializes and returns the receiver, a new instance of TextFieldCell, with *aString* as its text. The default text gray is NX_BLACK, and the default background gray is transparent (–1.0). Its font is set to the user's system font, and the font size is 12.0 point.

This method is the designated initializer for TextFieldCell. Override this method if you create a subclass of TextFieldCell that performs its own initialization. Note that TextFieldCell doesn't override Cell's **initIconCell:** designated initializer; your code shouldn't use that method to initialize an instance of TextFieldCell.

See also: – **init**

isBackgroundTransparent

– (BOOL)**isBackgroundTransparent**

Returns YES if the background of the TextFieldCell is transparent (that is, if the background gray is less than 0).

See also: – **setBackgroundTransparent:**, – **setBackgroundGray:**

isOpaque

– (BOOL)**isOpaque**

Returns YES if the TextFieldCell draws over every pixel in its frame. This will be true if the cell is bezeled, or if its background gray is not transparent.

See also: – **setBezeled:**, – **setBackgroundGray:**

read:

– **read:**(NXTypedStream *)*stream*

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

See also: – **write:**

setBackgroundColor:

– **setBackgroundColor:**(NXColor)*aColor*

Sets the background color for the TextFieldCell. Returns **self**.

See also: – **backgroundColor**, – **setBackgroundGray:**

setBackgroundGray:

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

Sets the gray level that will be used to draw the background. If *value* is less than 0.0, no background will be drawn. If the cell is editable, it should have a background gray greater than or equal to 0.0. Returns **self**.

See also: – **backgroundGray**, – **setBackgroundColor:**

setBackgroundTransparent:

– **setBackgroundTransparent:**(BOOL)*flag*

If *flag* is YES, sets the background gray of the TextFieldCell to transparent (a negative value); if NO, sets the background gray to NX_WHITE. Returns **self**.

See also: – **setBackgroundGray:**

setBezeled:

– **setBezeled:**(BOOL)*flag*

If flag is YES, the TextFieldCell is drawn with a bezel around the edge; if NO, nothing is drawn around the text. If the current background gray is transparent, it's changed to NX_WHITE. Bezeled transparent TextFieldCells look rather strange, but if you want to have one, invoke **setBackgroundGray:** with –1.0 *after* invoking **setBezeled:**.

See also: – **isBezeled** (Cell), – **setBackgroundGray:**

setTextAttributes:

– **setTextAttributes:***textObject*

Used to set the attributes of the field editor when editing the TextFieldCell (see the TextFieldCell class description). Sets the background and text colors or gray levels of *textObject* to those of the TextFieldCell, and returns *textObject*. *textObject* should respond to the messages **setBackgroundGray:**, **setBackgroundColor:**, **setTextGray:**, and **setTextColor:**. You rarely need to override this method; you never need to invoke it.

If the TextFieldCell is disabled, then *textObject*'s text color or gray level is brought toward the background's brightness level by 0.333. 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 color is black and the text color is red at 100% brightness, then the disabled text color would be red at 66.7% brightness.

Note that if the TextFieldCell has a transparent background, *textObject's* background gray isn't changed. Since a TextFieldCell's background is transparent by default, and the field editor's background could be any gray level or color (depending on where it was last used), this can cause ugly side effects. Editable TextFieldCells should use an opaque background whenever possible in order to avoid this.

See also: – **setTextGray:**, – **setBackgroundGray:**, – **setTextAttributes:** (Cell)

setTextColor:

– **setTextColor:**(NXColor)*aColor*

Sets the color used to draw the text. Returns **self**.

See also: – **textColor**, – **setTextGray:**

setTextGray:

– **setTextGray:**(float)*value*

Sets the gray level used to draw the text. Returns **self**.

See also: – **textGray**, – **setTextColor:**

textColor

– (NXColor)**textColor**

Returns the color used to draw the text. Returns **self**.

See also: – **setTextColor:**, – **textGray**

textGray

– (float)**textGray**

Returns the gray level used to draw the text. Returns **self**.

See also: – **setTextGray:**, – **textColor**

trackMouse:inRect:ofView:

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

Causes editing to occur, and increments the state of the TextFieldCell if its enabled and the mouse goes up in its frame. Returns YES if the mouse goes up in the TextFieldCell, NO otherwise.

See also: – **trackMouse:inRect:ofView:** (Cell)

write:

– **write:**(NXTypedStream *)*stream*

Writes the receiving TextFieldCell to the typed stream *stream*. Returns **self**.

See also: – **read:**

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. Any application that needs to display, print, or receive events must use View objects.

To be displayed, a View must be placed in a Window. All the Views within a 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 coordinate system). Both are represented by NXRect structures.

Instance Variables

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	The size and location of the View in its superview's coordinate system.
bounds	The size and location of the View in its own coordinate system.
superview	The View's parent in the view hierarchy.
subviews	A list of the View's immediate children in the view hierarchy.
window	The Window in which the View is displayed.
vFlags.noClip	YES if drawing isn't clipped to the frame.
vFlags.translatedDraw	YES if the bounds rectangle origin isn't (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 has changed since it was last 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::

Modifying the coordinate system	– boundsAngle – drawInSuperview – getBounds: – isFlipped – isRotatedFromBase – isRotatedOrScaledFromBase – rotate: – scale:: – setDrawOrigin:: – setDrawRotation: – setDrawSize:: – setFlipped: – translate::
Converting coordinates	– centerScanRect: – convertPoint:fromView: – convertPoint:toView: – convertPointFromSuperview: – convertPointToSuperview: – convertRect:fromView: – convertRect:toView: – convertRectFromSuperview: – convertRectToSuperview: – convertSize:fromView: – convertSize:toView:
Notifying ancestor Views	– descendantFlipped: – descendantFrameChanged: – notifyAncestorWhenFrameChanged: – notifyWhenFlipped: – suspendNotifyAncestorWhenFrameChanged:
Resizing subviews	– resizeSubviews: – setAutoresizeSubviews: – setAutosizing: – autosizing – superviewSizeChanged:
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: – shouldDelayWindowOrderingForEvent:

Dragging	– dragFile:fromRect:slideBack:event: – dragImage:at:offset:event:pasteboard:source:slideBack: – registerForDraggedTypes:count: – unregisterDraggedTypes
Printing	– printPSCode: – faxPSCode: – faxPSCode:toList:numberList:sendAt:wantsCover: wantsNotify:wantsHires:faxName: – copyPSCodeInside:to: – writePSCodeInside:to: – openSpoolFile: – spoolFile: – canPrintRIB
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**

This returns YES if an initial mouse-down event in the View—an event that causes the View's Window to become the key window—is sent to the View (through a **mouseDown:** message). If only those mouse-downs that occur when the View's Window is already key are sent, this returns NO (the default). The only way to change the default behavior is to implement this method in a View subclass.

addCursorRect:cursor:

– **addCursorRect:**(const NXRect *)*aRect* **cursor:***anNXCursor*

Creates a *cursor rectangle*, an area within the View that has its own cursor: When the user moves the mouse within the rectangle specified by *aRect*, the cursor object that the mouse controls changes to *anNXCursor*, which must be an NXCursor object. The rectangle is given in the View's coordinate system; however, the rectangle *isn't* automatically clipped to the View's frame—it's possible to create a cursor rectangle that extends beyond the View. You should also note that cursor rectangles don't work well in rotated Views.

You never invoke this method directly from your application. It should only be used as part of the implementation of the **resetCursorRects** method.

Returns **self**.

See also: – **resetCursorRects**

addSubview:

– **addSubview:***aView*

Adds *aView* to the View's list of subviews such that new subview will be displayed on top of its siblings. The receiving View is also made *aView*'s next responder. Returns *aView* (or **nil** if it isn't a View).

See also: – **addSubview::relativeTo:**, – **subviews**, – **removeFromSuperview**, – **setNextResponder:** (Responder)

addSubview::relativeTo:

– **addSubview:***aView*
:(int)*place*
relativeTo:*otherView*

Injects *aView* into the receiving View's list of subviews, such that it will be displayed immediately above or below *otherView*, as *place* is NX_ABOVE or NX_BELOW. If *otherView* is **nil** (or isn't in the subview list), *aView* is added above or below all its siblings. Returns *aView* (or **nil** if it isn't a View).

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)

autosizing

– (unsigned int)**autosizing**

Returns the View's autosizing mask. The mask is used to determine how the View is automatically resized when its superview is resized. For the mask to have an effect, the superview must be set to resize its subviews; this is done through the **setAutoresizeSubviews:** method. The autosizing masks are listed under the **setAutosizing:** method.

See also: – **setAutosizing:**, – **setAutoresizeSubviews:**

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)

canPrintRIB

– (BOOL)**canPrintRIB**

Indicates whether the View can print RIB files.

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* is **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* is **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 View's coordinate system to that of its superview. Returns **self**.

See also: – **convertPointFromSuperview:**, – **convertPoint:fromView:**

convertRect:fromView:

– **convertRect:**(NXRect *)*aRect* **fromView:***aView*

Converts *aRect* from *aView*'s coordinate system to the coordinate system of the receiving View. Both *aView* and the receiving View must belong to the same Window. Returns **self**.

convertRect:toView:

– **convertRect:**(NXRect *)*aRect* **toView:***aView*

Converts *aRect* 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**.

convertRectFromSuperview:

– **convertRectFromSuperview:**(NXRect *)*aRect*

Converts *aRect* 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 *aRect* 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* 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* 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 the View's cursor rectangles. You never invoke this method directly; it's invoked automatically before the View's cursor rectangles are reset. Returns **self**.

See also: – **resetCursorRects**, – **discardCursorRects** (Window)

display

– **display**

Displays the View and its subviews. Returns **self**. This method is equivalent to:

```
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:

```
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. The third argument, *clipFlag*, is the same as the third argument to **display:::**.

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 YES (the default) if the drawing that's generated by the View is clipped to the View's frame; otherwise returns NO.

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. This method only makes sense when invoked from within an implementation of the **mouseDown:** method. The arguments are:

- *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; 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* is the mouse-down event record (or a copy).

This method returns **self** if the View successfully initiated the file dragging session; otherwise it returns **nil**.

See also: – **dragImage:at:offset:event:pasteboard:source:slideBack:**

dragImage:at:offset:event:pasteboard:source:slideBack:

– **dragImage:***anImage*
at:(NXPoint *)*location*
offset:(NXPoint *)*mouseOffset*
event:(NXEvent *)*theMouseDown*
pasteboard:(Pasteboard *)*pboard*
source:*sourceObject*
slideBack:(BOOL)*slideFlag*

Instigates an image-dragging session. This method only makes sense when invoked from within an implementation of the **mouseDown:** method. The arguments are

- *anImage* is the NXImage (contained within the View) that's being dragged.
- *location* is the NXImage's origin in the View's coordinate system.
- *mouseOffset* gives the mouse's current location relative to the mouse-down location.
- *theMouseDown* is the mouse-down that started the whole thing going (see below).
- *pboard* is the Pasteboard that holds the data that the NXImage represents (see below).
- *sourceObject* is the object that receives NXDraggingSource messages.
- *slideFlag* determines whether the NXImage should slide back if it's rejected.

Before invoking this method, the View must place the data that's being dragged on the drag pasteboard. To do this, it must get the pasteboard, declare the type of data that it's placing, and then place the data:

```
/* You always use the NXDragPboard pasteboard when dragging. */
Pasteboard *pboard = [Pasteboard newName:NXDragPboard];

/* Declare the type of data and place it on the pasteboard. */
[pboard declareTypes:... num:... owner:...];
[pboard writeType:... data:... length:...];

/* Now invoke dragImage:.*/
[self dragImage:... at:... offset:... event:...
      pasteboard:pboard source:... slideBack:...];
```

This method returns **self** if the View successfully initiated the file dragging session; otherwise it returns **nil**.

Warning: If you ask for events inside the **mouseDown:** method before invoking this method (if, for example, you're making sure that the image is really being dragged before initiating a dragging session), you must copy the mouse-down event *before* asking for more events. You then pass the copy as the argument to the **event:** keyword of this method.

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's 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 **endProlog**ue. 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 Additions)

faxPSCode:toList:numberList:sendAt:wantsCover:wantsNotify: wantsHires:faxName:

– **faxPSCode:***sender*
toList:(const char *const *)*names*
numberList:(const char *const *)*numbers*
sendAt:(time_t)*time*
wantsCover:(BOOL)*coverFlag*
wantsNotify:(BOOL)*notifyFlag*
wantsHires:(BOOL)*hiresFlag*
faxName:(const char *)*string*

Sets up a fax session according to the arguments, and then faxes the View (and all its subviews).

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*

Returns the View's nearest descendant (including itself) that has the given tag, or **nil** if no matching tag was found.

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 (as set through **setOpaque:**). Returns YES if the View guarantees that it will completely cover the area within its frame when it draws itself; otherwise returns NO.

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*. *aPoint* and *aRect* must be expressed in the same coordinate system.

You should never use the **NXPointInRect()** function as a substitute for this method.

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 View's closest opaque ancestor (including the receiving View itself).

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, 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 returns NO, 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 Additions)

read:

– **read:**(NXTypedStream *)*stream*

Reads the View and its subviews from the typed stream *stream*. Returns **self**.

registerForDraggedTypes:count:

– **registerForDraggedTypes:**(const char *const *)*pbTypes* **count:**(int)*count*

Registers the Pasteboard types that the View will accept in an image-dragging session. *pbTypes* is a pointer to an array of the types; *count* is the number of elements in the array. Returns **self**.

Keep in mind that the values in the first argument are Pasteboard types, *not* file extensions (you can't register for specific file extensions). For example, the following registers a View as accepting files:

```
const char *fileType[] = {NXFilenamePboardType};
[aView registerForDraggedTypes:fileType count:1];
```

Note: Registering a View for dragged types automatically makes it a candidate destination object during a dragging session. As such, it must implement some or all of the NXDraggingDestination protocol methods. As a convenience, View provides default implementations of these methods. See the NXDraggingDestination protocol description for details.

See also: – **unregisterDraggedTypes**

removeCursorRect:cursor:

– **removeCursorRect:**(const NXRect *)*aRect* **cursor:***anNXCursor*

Removes a cursor rectangle from the View. *aRect* and *anNXCursor* must match the values that were specified when the cursor rectangle was added (through **addCursorRect:cursor:**).

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 isn't 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

Resets the View's cursor rectangles. Each View subclass that wants to include cursor rectangles—areas in which the cursor is changed—must implement this method. The implementation must contain invocations of **addCursorRect:cursor:**, the method that defines the cursor rectangles and associates them with particular NXCursor objects. The View must clip the cursor rectangles that it adds to ensure that they don't overlap the visible rectangle. For example:

```
- resetCursorRects
{
   NXRect visible;

   if ([self getVisibleRect:&visible])
      [self addCursorRect:&visible cursor:theCursor];
   return self.
}
```

The default implementation does nothing. The value returned by this method is ignored.

You never invoke this method directly; it's invoked automatically when the View's Window frame changes, or when the Window receives an **invalidateCursorRectsForView:** message. Note that this method isn't invoked when the View's frame changes (unless it changed because its Window was resized). If your application changes a View's frame programmatically, through **sizeBy::** or **moveBy::** for example, you should follow the frame-changing message with an **invalidateCursorRectsForView:** message, as shown below:

```
/* Shrink the View's frame. */
[aView sizeBy:-10.0 :-10.0];

/* Tell the Window that the View's cursor rects may have changed. */
[[aView window] invalidateCursorRectsForView:aView];

/* Redisplay the Window. */
[[aView window] display];
```

Invocations of this method aren't cumulative; before a **resetCursorRects** message is sent to a particular View, the View's existing cursor rectangles are automatically discarded.

See also: – **addCursorRect:**, – **invalidateCursorRectsForView:** (Window)

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*

Shifts the View's coordinate system so that (*x*, *y*) corresponds to the same point as the View's frame rectangle origin. If the View's 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 coordinate system 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 coordinate system so that *width* and *height* define the size of the View's frame rectangle in its own 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*

Sets the direction of the View's y-axis. If *flag* is YES, the View's origin will be its upper left corner, and coordinate values will increase towards the bottom of the View. If *flag* is NO, the origin is the bottom left corner and values increase to the top; this is the default configuration for a View.

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 doesn'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 a View can guarantee that it will completely cover the area within its frame (with opaque paint), it should send itself a **setOpaque:**YES message (typically, as part of its initialization). Opaque Views make drawing in the view hierarchy more efficient. Returns **self**.

See also: – **isOpaque**, – **opaqueAncestor**, – **displayFromOpaqueAncestor:::**

shouldDelayWindowOrderingForEvent:

– (BOOL)**shouldDelayWindowOrderingForEvent:**(NXEvent *)*anEvent*

Returns YES if the normal Window ordering and activation mechanism should be delayed until the next mouse-up event. You never inovke this method directly; it's invoked automatically for each mouse-down that's directed at the View. The default implementation returns NO.

A View subclass that contains draggable images should implement this to return YES (perhaps predicating the decision on the data in *anEvent*, the event record for the

mouse-down itself). This allows the user to click on a draggable image without bringing the View's Window to the front or making its application active. Note that this method doesn't prevent this ordering and activation from occurring, it simply puts it off until the user releases the mouse. To cause the ordering and activation to be skipped when the mouse is released, the View should send a **preventWindowOrdering** message to the Application object from within its implementation of **mouseDown:**. The **preventWindowOrdering** message is sent automatically by View's **dragImage:...** method—in other words, ordering and activation is prevented if the user actually drags the clicked-on item.

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 never modify this List directly; use **addSubview:** and **removeFromSuperview** to add and remove Views from the View hierarchy. If the View has no subviews (and never did), **nil** is returned. If it had subviews that have all since been removed, an empty List is returned.

See also: – **superview**, – **addSubview:**, – **removeFromSuperview**

superview

– **superview**

Returns the View's superview. If the View hasn't a superview, **nil** is returned. When applying this method recursively, you should check the return value against the content View of the View's Window to avoid flying off the top of the View hierarchy.

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:::**

unregisterDraggedTypes

– **unregisterDraggedTypes**

Unregisters the View as a possible recipient of dragged-images.

See also: – **registerForDraggedTypes:count:**

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 View's Window.

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 View and its subviews to the typed stream *stream*. Returns **self**.

writePSCodeInside:to:

– **writePSCodeInside:**(const NXRect *)*aRect* **to:***pasteboard*

Copies the portions of the View and its subviews that fall inside *aRect* and places the copy on the given pasteboard. Returns **self**.

Window

Inherits From: Responder : Object

Declared In: appkit/Window.h

Class Description

The Window class defines objects that manage and coordinate the windows that an application displays on the screen. A single Window object corresponds to, at most, one window. The two principle functions of a Window are to provide an area in which Views can be placed, and to accept and distribute, to the appropriate Views, events that the user instigates through actions on the mouse and keyboard.

Rectangles, Views, and the View Hierarchy

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. The frame rectangle establishes the Window's *base coordinate system.* This coordinate system is always aligned with and is measured in the same increments as the screen coordinate system (in other words, the base coordinate system can't be rotated or scaled). The origin of a base coordinate system is the bottom left corner of the Window's frame rectangle.

You create a Window (through one of the **init:...** methods) by specifying, among other attributes, the size and location of its content rectangle. The frame rectangle is derived from the dimensions of the content rectangle.

When it's created, a Window automatically creates two Views: an opaque *frame view* that fills the frame rectangle and draws the border, title bar, resize bar, and background, and a transparent *content view* that fills the content area. The frame view is a private object that your application can't access directly. The content view is the "highest" accessible View in the Window; you can replace the content view with a View of your own creation through Window's **setContentView:** method.

You add other Views to the Window by declaring each to be a subview of the content view, or a subview of one of the content view's subviews, and so on, through View's **addSubview:** method. This tree of Views is called the Window's *view hierarchy.* When a

Window is told to display itself, it does so by sending View-displaying messages to each object in its view hierarchy. Because displaying is carried out in a determined order, the content view (which is drawn first) may be wholly or partially obscured by its subviews, and these subviews may be obscured by their subviews (and so on).

Event Handling

Mouse and keyboard events that are directed at a Window are automatically forwarded to the object, however, a Window receives keyboard events only if it's the *key window.* If the event affects the Window directly—resizing or moving it, for example—the Window performs the appropriate operation itself and sends messages to its delegate informing it of its intentions, thus allowing your application to intercede. Events that are directed at specific Views within the Window are forwarded by the Window to the View.

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, the first responder is sent action messages that have a user-selected target (a **nil** target in program code). The Window continually updates the first responder in response to the user's mouse actions.

Each Window provides a *field editor*, a Text object that handles small-scale text-editing chores. The field editor can be used by the Window's first responder to edit the text that it displays. The **getFieldEditor:for:** method returns a Window's field editor.

Instance Variables

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 frame rectangle in screen coordinates.
contentView	The View that fills the Window's content area.
delegate	The object that receives notification messages.
firstResponder	The Responder object that receives keyboard events and untargeted action messages sent to the Window.
lastLeftHit	The View in the Window's view hierarchy that most recently received a left mouse-down event.
lastRightHit	The View in the Window's view hierarchy that most recently received a right mouse-down event.
counterpart	The Window's miniwindow, or, if the Window is a miniwindow, the Window it stands for.
fieldEditor	The Text object that displays and edits text for the Window.
winEventMask	The events the Window can receive from the Window Server.
windowNum	An integer that identifies the Window Server window device that corresponds to this Window.
backgroundGray	The shade of gray that fills the Window's background.

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	A mask that indicates whether the Window has a close button and miniaturize button.
wFlags.visible	True if the Window is on-screen (if it's in the screen list).
wFlags.isMainWindow	True if the Window is the main window.
wFlags.isKeyWindow	True if 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 is deactivated.
wFlags.dontFreeWhenClosed	True if the Window is not to be freed when closed.
wFlags.oneShot	True if the Window Server should free the window device for this object when the Window is removed from the screen.
wFlags2.deferred	True if the Window Server shouldn't create a window device for this object until it's placed on-screen.
wFlags2.docEdited	True if the close button indicates that a displayed document has been edited but not saved.
wFlags2.dynamicDepthLimit	True if the Window has a depth limit that can change to match the depth of the screen.

Method Types

Initializing a new Window instance

- init
- initContent:style:backing:buttonMask:defer:
- initContent:style:backing:buttonMask: defer:screen:

Freeing a Window object

- free

Computing frame and content rectangles

+ getFrameRect:forContentRect:style:
+ getContentRect:forFrameRect:style:
+ minFrameWidth:forStyle:buttonMask:

Accessing the frame rectangle	– getFrame: – getFrame:andScreen: – setFrameUsingName: – saveFrameUsingName: + removeFrameUsingName: – setFrameAutosaveName: – frameAutosaveName – setFrameFromString: – saveFrameToString:
Accessing the content view	– setContentView: – contentView
Querying Window attributes	– windowNum – buttonMask – style – worksWhenModal
Window graphics	– setTitle: – setTitleAsFilename: – title – setBackgroundColor: – backgroundColor – setBackgroundGray: – backgroundGray
Window device attributes	– setBackingType: – backingType – setOneShot: – isOneShot – setFreeWhenClosed:
The miniwindow	– counterpart – setMiniwindowIcon: – setMiniwindowImage: – setMiniwindowTitle: – miniwindowIcon – miniwindowImage – miniwindowTitle
The field editor	– endEditingFor: – getFieldEditor:for:

Window status	– makeKeyWindow – makeKeyAndOrderFront: – becomeKeyWindow – isKeyWindow – resignKeyWindow – canBecomeKeyWindow – becomeMainWindow – isMainWindow – resignMainWindow – canBecomeMainWindow
Moving and resizing	– moveTo:: – moveTo::screen: – moveTopLeftTo:: – moveTopLeftTo::screen: – dragFrom::eventNum: – constrainFrameRect:toScreen: – placeWindow: – placeWindow:screen: – placeWindowAndDisplay: – sizeWindow:: – setMinSize: – setMaxSize: – getMinSize: – getMaxSize: – resizeFlags – center
Ordering on and off screen	– makeKeyAndOrderFront: – orderFront: – orderBack: – orderOut: – orderWindow:relativeTo: – orderFrontRegardless – isVisible – setHideOnDeactivate: – doesHideOnDeactivate
Converting coordinates	– convertBaseToScreen: – convertScreenToBase:

Managing display	– display – displayIfNeeded – disableDisplay – isDisplayEnabled – reenableDisplay – flushWindow – flushWindowIfNeeded – disableFlushWindow – reenableFlushWindow – isFlushWindowDisabled – displayBorder – useOptimizedDrawing: – update
Screens and Window depths	– screen – bestScreen + defaultDepthLimit – setDepthLimit: – depthLimit – setDynamicDepthLimit: – hasDynamicDepthLimit – canStoreColor
Graphics state objects	– gState
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: – 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: – setAvoidsActivation: – avoidsActivation
Dragging	– registerForDraggedTypes:count: – unregisterDraggedTypes – dragImage:at:offset:event:pasteboard:source:slideBack:
Services and Windows menu support	– validRequestorForSendType:andReturnType: – setExcludedFromWindowsMenu: – isExcludedFromWindowsMenu
Assigning a delegate	– setDelegate: – delegate
Printing	– printPSCode: – smartPrintPSCode: – faxPSCode: – smartFaxPSCode: – openSpoolFile: – spoolFile: – copyPSCodeInside:to: – knowsPagesFirst:last: – getRect:forPage: – placePrintRect:offset: – heightAdjustLimit – widthAdjustLimit – beginPSOutput – endPSOutput – beginPrologueBBox:creationDate: createdBy:fonts:forWhom:pages:title: – endHeaderComments

	– endPrologue – beginSetup – endSetup – beginPage:label:bBox:fonts: – endPage – beginPageSetupRect:placement: – endPageSetup – beginTrailer – endTrailer
Archiving	– read: – write: – awake

Class Methods

defaultDepthLimit

+ (NXWindowDepth)**defaultDepthLimit**

Returns the default depth limit for instances of Window. This will be the smaller of:

- The depth of the deepest display device available to the Window Server.
- The depth set for the application by the NXWindowDepthLimit parameter.

The value returned will be one of these NXWindowDepth values:

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 and returns, in *content*, the content rectangle for a Window with the given frame rectangle and style. Both *content* and *frame* are in screen coordinates. See the **style** method for a list of acceptable style values. Returns **self**.

See also: + **getFrameRect:forContentRect:style:**

getFrameRect:forContentRect:style:

+ **getFrameRect:**(NXRect *)*frame*
  forContentRect:(const NXRect *)*content*
  style:(int)*aStyle*

Calculates and returns, in *frame*, the content rectangle for a Window with the given frame rectangle and style. Both *frame* and *content* are in screen coordinates. See the **style** method for a list of acceptable style values. Returns **self**.

See also: + **getContentRect:forFrameRect:style:**

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. See the **style** and **buttonMask** methods for lists of acceptable style and button mask values.

removeFrameUsingName:

+ (void)**removeFrameUsingName:**(const char *)*name*

Removes the frame data named *name* from the application's defaults.

See also: – **setFrameUsingName:**, – **setFrameAutosaveName:**

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**. *aRect*, which is taken in the Window's base coordinate system, must lie within the Window's content rectangle. If it doesn't, the cursor rectangle isn't added and **nil** is returned.

You typically add cursor rectangles to View objects (through View's **addCursorRect:cursor:** method) rather than to Windows.

See also: – **addCursorRect:cursor:** (View)

addToEventMask:

– (int)**addToEventMask:**(int)*newEvents*

Adds *newEvents* to the Window's current event mask and returns the original event mask. 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 **setEventMask:** for a list of event mask constants.

See also: – **setEventMask:**, – **eventMask**, – **removeFromEventMask:**

avoidsActivation

– (BOOL)**avoidsActivation**

Returns YES if the Window's application doesn't become active when the user clicks in the Window's content area. The default is NO. Note that clicking on the title bar will always activate the Window's application.

See also: – **setAvoidsActivation:**

awake

– **awake**

You never invoke this method directly; it's invoked automatically after the Window has been read from an archive file.

See also: – **read:**

backgroundColor

– (NXColor)**backgroundColor**

Returns the color of the Window's background when the object is displayed on a color screen. The default is the color equivalent of NX_LTGRAY.

See also: – **setBackgroundColor:**, – **setBackgroundGray:**

backgroundGray

– (float)**backgroundGray**

Returns the shade of gray of the Window's background when the object is displayed on a monochrome screen. The default is NX_LTGRAY.

See also: – **setBackgroundGray:**, – **setBackgroundColor:**

backingType

– (int)**backingType**

Returns the Window's backing type as one of the following constants:

NX_BUFFERED
NX_RETAINED
NX_NONRETAINED

See also: – **setBackingType:**

becomeKeyWindow

– **becomeKeyWindow**

You never invoke this method; it's invoked automatically when the Window becomes the key window. The method sends **becomeKeyWindow** to the Window's first responder, and **sends windowDidBecomeKey:** to the Window's delegate (if the respective objects can respond). Returns **self**.

See also: – **makeKeyWindow**, – **makeKeyAndOrderFront:**

becomeMainWindow

– **becomeMainWindow**

You never invoke this method; it's invoked automatically when the Window becomes the main window. The method sends **windowDidBecomeMain:** to the Window's delegate (if the delegate can respond). Returns **self**.

See also: – **makeKeyWindow**, – **makeKeyAndOrderFront:**

beginPage:label:bBox:fonts:

– **beginPage:**(int)*ordinalNum*
label:(const char *)*aString*
bBox:(const NXRect *)*pageRect*
fonts:(const char *)*fontNames*

Writes a PostScript page separator by forwarding the **beginPage:...** message to the Window's frame view. You never invoke this method directly; it's invoked automatically when printing or faxing the Window.

See also: – **beginPage:labelbBox:fonts:** (View)

beginPageSetupRect:placement:

– **beginPageSetupRect:**(const NXRect *)*aRect*
placement:(const NXPoint *)*location*

Writes the start of a PostScript page-setup section by forwarding the **beginPageSetupRect:placement:** message to the Window's frame view. You never invoke this method directly; it's invoked automatically when printing or faxing the Window.

See also: – **beginPageSetupRect:placement:** (View)

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 PostScript prolog section by forwarding the **beginPrologueBbox:...** message to the Window's frame view. You never invoke this method directly; it's invoked automatically when printing or faxing the Window.

See also: – **beginPrologueBBox:..** (View)

beginPSOutput

– **beginPSOutput**

Prepares the Window (and the application environment) for printing or faxing by forwarding the **beginPSOutput** message to the Window's frame view. You never invoke this method directly; it's invoked automatically when printing or faxing the Window.

See also: – **beginPSOutput** (View)

beginSetup

– **beginSetup**

Writes the start of a PostScript document-setup section by forwarding the **beginSetup** message to the Window's frame view. You never invoke this method directly; it's invoked automatically when printing or faxing the Window.

See also: – **beginSetup** (View)

beginTrailer

– **beginTrailer**

Writes the start of a PostScript document-trailer section by forwarding the **beginTrailer** message to the Window's frame view. You never invoke this method directly; it's invoked automatically when printing or faxing the Window.

See also: – **beginTrailer** (View)

bestScreen

– (const NXScreen *)**bestScreen**

Returns a pointer to the deepest screen that the Window is on, or NULL if the Window is currently off-screen.

See also: – **screen**, – **colorScreen** (Application)

buttonMask

– (int)**buttonMask**

Returns a mask that indicates which buttons appear in the Window's title bar. The return value may include one or both of these constants:

NX_CLOSEBUTTONMASK
NX_MINIATURIZEBUTTONMASK

The button mask is set when the Window is initialized and is, thereafter, immutable.

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

canBecomeKeyWindow

– (BOOL)**canBecomeKeyWindow**

Returns YES if the Window can be made the key window, and NO if it can't. This method is consulted when the Window tries to become the key window; the attempt is thwarted if this method returns NO.

See also: – **isKeyWindow**, – **makeKeyWindow**

canBecomeMainWindow

– (BOOL)**canBecomeMainWindow**

Returns YES if the Window can be made the main window, and NO if it can't. This method is consulted when the Window tries to become the main window; the attempt is thwarted if this method returns NO.

See also: – **isMainWindow**, – **makeKeyWindow**

canStoreColor

– (BOOL)**canStoreColor**

Returns YES if the Window has a depth limit that allows 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: The Window is placed dead-center horizontally and placed somewhat above center vertically. Such a placement is consider to carry a certain immediacy and importance, visually. You typically use this method to place a Window—most likely an attention Panel—where the user can't miss it. This method is invoked automatically when a Panel is placed on the screen by Application's **runModalFor:** method. Returns **self**.

close

– **close**

Removes the Window from the screen. If the Window is set to be freed when it's closed (the default), a **free** message is sent to the object.

Normally, this method is invoked by the Application Kit when the user clicks the Window's close button. Note that this method *doesn't* cause **windowWillClose:** to be sent to the Window's delegate (the message *is* sent when the user clicks the close button). You can induce an invocation of the delegate method by simulating the user's action through the **performClose:** method.

Returns **nil**.

See also: – **performClose:**, – **setFreeWhenClosed:**

commandKey:

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

Responds to the Command key-down event passed as *theEvent*. You never invoke this method directly; the Application object, upon receiving a Command key-down event, sends a **commandKey:** message to each Window in the Window list until one of them returns YES (signifying that the event was recognized and handled). The default implementation of this method returns NO—instances of Window can't handle these events. (By contrast, Panels can.)

You can create your own subclass of Window that responds to Command key-down events. A typical subclass implementation of this method passes a **performKeyEquivalent:** message down the view hierarchy:

```
- (BOOL)commandKey:(NXEvent *)theEvent
{
    if ( [contentView performKeyEquivalent:theEvent] )
        return YES;
    else
        return NO;
}
```

See also: – **performKeyEquivalent:** (View), – **commandKey:** (Panel)

constrainFrameRect:toScreen:

– (BOOL)**constrainFrameRect:**(NXRect *)*theFrame*
toScreen:(const NXScreen *)*screen*

Modifies the rectangle pointed to by *theFrame* such that its top edge lies on the given screen. If the Window is resizable, the rectangle's height is adjusted to bring the bottom edge onto the screen as well. The rectangle's width and horizontal location are unaffected. You shouldn't need to invoke this method yourself; it's invoked automatically (and the modified frame is used to locate and set the size of the Window) whenever a titled Window is placed on-screen or resized through **sizeWindow::**.

You can override this method to prevent a particular Window from being constrained, or to constrain it differently. The unconstrained frame rectangle is pointed to by *theFrame*; the screen it wants to lie on is pointed to by *screen*. If your method modifies the rectangle, it should return YES; otherwise, it should return NO.

contentView

– **contentView**

Returns the Window's **content view**, the highest accessible View object in the Window's view hierarchy.

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. 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. Returns **self**.

See also: – **convertBaseToScreen:**

copyPSCodeInside:to:

– **copyPSCodeInside:**(const NXRect *)*rect* **to:**(NXStream *)*stream*

Generates PostScript code, in the manner of **printPSCode:**, 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*.

Returns **self** (unless an exception is raised).

See also: – **printPSCode:**, – **faxPSCode:**

counterpart

– **counterpart**

Returns the Window's miniwindow or, if this Window *is* a miniwindow, the Window that it represents. You can't set a Window's counterpart directly; a corresponding miniwindow is created automatically the first time the Window is miniaturized. If the Window has not yet been miniaturized, this method will return **nil**.

See also: – **setMiniwindowImage:**, – **setMiniwindowTitle:**

delegate

– **delegate**

Returns the Window's delegate, or **nil** if it doesn't have one.

See also: – **setDelegate:**

deminiaturize:

– **deminiaturize:***sender*

Deminiaturizes the Window (which should be a miniwindow). You rarely need to invoke this method; it's invoked automatically when a Window is deminiaturized by the user (by double-clicking a miniwindow, or by choosing the Arrange in Front item in the Windows menu). However, if you feel compelled to deminiaturize a Window programmatically, you should note that the **deminiaturize** message is sent to the miniwindow, *not* the original Window. The value passed as *sender* is ignored. Returns **self**.

See also: – **miniaturize:**

depthLimit

– (NXWindowDepth)**depthLimit**

Returns the depth limit of the Window as one of the following values:

NX_DefaultDepth
NX_TwoBitGrayDepth
NX_EightBitGrayDepth
NX_TwelveBitRGBDepth
NX_TwentyFourBitRGBDepth

If the return value is NX_DefaultDepth, you can find out the actual depth limit 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

Disables View's display methods, thus preventing the Views in the Window's view hierarchy from being displayed (note, however, that this doesn't disable Window's **display** method). 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 or until a **display** message is sent to the Window.

Returns **self**.

See also: **– reenableDisplay**, **– isDisplayEnabled**, **– display**, **– 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 remove 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 *trackNum* and returns **self**. The tag was assigned when the tracking rectangle was created.

See also: – **setTrackingRect:inside:owner:tag:left:right:**

display

– **display**

Passes a **display** message down the Window's view hierarchy, thus redrawing all Views within the Window, including the border, resize bar, and title bar. If displaying is disabled for 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**. You rarely need to invoke this method yourself; a Window's border is automatically displayed when any of the elements therein are changed—when the Window is resized or its title is changed, for example.

See also: – **display**

displayIfNeeded

– **displayIfNeeded**

Sends a **displayIfNeeded** message down the Window's view hierarchy, thus redrawing all Views that need to be displayed, including the Window's border, title bar, and resize bar. This method is useful when you want to disable displaying in the Window, modify some number of Views, and then display only the ones that were modified. Note that this method, unlike **display**, doesn't reenable display if it's currently disabled. Returns **self**.

See also: – **display**, – **displayIfNeeded** (View), – **setNeedsDisplay:** (View), – **update** (View)

doesHideOnDeactivate

– (BOOL)**doesHideOnDeactivate**

Returns YES if the Window will be removed from the screen when its application is deactivated, and NO if it will remain on-screen.

See also: – **setHideOnDeactivate:**

dragFrom::eventNum:

– **dragFrom:**(float)*x*
:(float)*y*
eventNum:(int)*num*

Lets the user drag a Window from a location other than the title bar.

Warning: This method has nothing to do with the image-dragging mechanism provided by the **dragImage:...** method and related protocols.

Normally, a Window can only be dragged by its title bar (if it has one). To provide some other draggable area in the Window, you design a View that invokes this method when it receives a mouse-down event. 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 arguments should be taken from the mouse-down event record. The following example shows an implementation of **mouseDown:** that would allow the user to drag the Window by clicking anywhere in the View:

```
- mouseDown:(NXEvent *)theEvent
{
    [window dragFrom:theEvent->location.x :theEvent->location.y
                eventNum:theEvent->data.mouse.eventNum];
    return self;
}
```

The dragging itself is performed as usual: The View that invoked this method won't receive the subsequent mouse-dragged and mouse-up events, they're intercepted and applied directly to change the Window's location.

Returns **self**.

See also: – **moveTo::**

dragImage:at:offset:event:pasteboard:source:slideBack:

– **dragImage:***anImage*
at:(NXPoint *)*location*
offset:(NXPoint *)*initialOffset*
event:(NXEvent *)*event*
pasteboard:(Pasteboard *)*pboard*
source:*sourceObject*
slideBack:(BOOL)*slideFlag*

Instigates an image-dragging session. You never invoke this method directly from your application; it can only be invoked from within a View's implementation of the **mouseDown:** method. Furthermore, View also implements the **dragImage:...** method; you typically instigate an image-dragging session by sending this message to a View, rather than a Window. The two methods are identical except for the interpretation of the *location* argument: In Window's implementation, *location* is taken in the base coordinate system. See the description of this method in the View class for the meanings of the other arguments.

See also: – **dragImage:at:offset:event:pasteboard:source:slideBack:** (View)

enableCursorRects

– **enableCursorRects**

Reenables cursor rectangle management. Returns **self**.

See also: – **disableCursorRects**

endEditingFor:

– **endEditingFor:***anObject*

Prepares the Window's field editor for a new editing assignment and returns **self**. The argument is ignored by Window's default implementation.

If the field editor is the first responder, it resigns that status, passing it to the Window (even if the field editor refuses to resign). This forces a **textDidEnd:endChar:** message to be sent to the field editor's delegate. The field editor is then removed from the view hierarchy and its delegate is set to **nil**.

To conditionally end editing, first try to make the Window the first responder:

```
if ([myWindow makeFirstResponder:myWindow]) {
    [myWindow endEditingFor:nil];
    . . .
}
```

This is the preferred way to verify all fields when an OK button is pressed in a panel, for example.

See also: **– getFieldEditor:for:**

endHeaderComments

– endHeaderComments

Writes the end of a PostScript comment section by forwarding the **endHeaderComments** message to the Window's frame view. You never invoke this method directly; it's invoked automatically when printing or faxing the Window.

See also: **– endHeaderComments** (View)

endPage

– endPage

Writes the end of a PostScript page separator by forwarding the **endPage** message to the Window's frame view. You never invoke this method directly; it's invoked automatically when printing or faxing the Window.

See also: **– endPage** (View)

endPageSetup

– endPageSetup

Writes the end of a PostScript page-setup section by forwarding the **endPageSetup** message to the Window's frame view. You never invoke this method directly; it's invoked automatically when printing or faxing the Window.

See also: **– endPageSetup** (View)

endPrologue

– **endPrologue**

Writes the end of a PostScript prolog section by forwarding the **endPrologue** message to the Window's frame view. You never invoke this method directly; it's invoked automatically when printing or faxing the Window.

See also: – **endPrologue** (View)

endPSOutput

– **endPSOutput**

Declares that printing or faxing is finished by forwarding the endPSOutput to the Window's frame view. You never invoke this method directly; it's invoked automatically when printing or faxing the Window.

See also: – **endPSOutput** (View)

endSetup

– **endSetup**

Writes the end of a PostScript document-setup section by forwarding the **endSetup** message to the Window's frame view. You never invoke this method directly; it's invoked automatically when printing or faxing the Window.

See also: – **endSetup** (View)

endTrailer

– **endTrailer**

Writes the end of a PostScript document-trailer section by forwarding the **endTrailer** message to the Window's frame view. You never invoke this method directly; it's invoked automatically when printing or faxing the Window.

See also: – **endTrailer** (View)

eventMask

– (int)**eventMask**

Returns the current event mask for the Window. See **setEventMask:** for a list of the possible contents of the mask.

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 previous settings of the panel.

See also: – **smartFaxPSCode:**, – **printPSCode:**,
– **shouldRunPrintPanel:** (Object Additions)

firstResponder

– **firstResponder**

Returns the Window's first responder.

See also: – **makeFirstResponder:**, – **acceptsFirstResponder** (Responder)

flushWindow

– **flushWindow**

if the Window is buffered and flushing hasn't been disabled by **disableFlushWindow**, this flushes the off-screen buffer to the screen. This method is automatically invoked when you send a display message to a Window or View. However, it has no effect if the display is being directed to a printer or other device, rather than to the screen. Returns **self**.

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

flushWindowIfNeeded

– **flushWindowIfNeeded**

Flushes the Window's off-screen buffer to the screen, provided that:

- The Window is a buffered window
- Flushing isn't currently disabled
- Some previous **flushWindow** messages had no effect because flushing was disabled

You should use this method, rather than **flushWindow**, to flush a Window after flushing has been reenabled. Returns **self**.

See also: – **flushWindow**, – **disableFlushWindow**, – **reenableFlushWindow**

frameAutosaveName

– (const char *)**frameAutosaveName**

Returns the name that's used to automatically save the Window's frame rectangle data in the defaults system, as set through **setFrameAutosaveName:**. If the Window has an autosave name, it's frame data is written as a default whenever the frame rectangle changes.

See also: – **setFrameAutosaveName:**

free

– **free**

Deallocates memory for the Window object and all that it surveys. This includes the Views in its view hierarchy, its instance variables (including the field editor), and the Window Server window device that it's associated with.

getFieldEditor:for:

– **getFieldEditor:**(BOOL)*flag* **for:***anObject*

Returns the field editor, the Window's communal Text object. The field editor is provided as a convenience and can be used however your application sees fit. Typically, the field editor is used by simple text-bearing objects—for example, a TextField object uses its Window's field editor to display and manipulate text. The field editor can be shared by any number of objects and so its state may be constantly changing. Therefore, it shouldn't be used to display text that demands sophisticated Text object preparation (for this you should create a dedicated Text object).

A freshly created Window doesn't have a field editor; the only way to create a field editor is to invoke this method with a *flag* value of YES. After a field editor has been created for a Window, the *flag* argument is ignored.

The Window's delegate can supply the object that this method returns as the return value of the **windowWillReturnFieldEditor:toObject:** delegate message (the Window is passed as the first argument, *anObject* is passed as the second). However, note the following:

- If the Window's delegate is *anObject*, **windowWillReturnFieldEditor:toObject:** isn't sent.
- The object returned by the delegate method *doesn't* become the Window's field editor.

If this method returns a non-**nil** value, it should be followed by an invocation of Window's **endEditingFor:** method before the field editor is actually used.

See also: – **endEditingFor:**

getFrame:

– **getFrame:**(NXRect *)*theRect*

Returns the Window's frame rectangle by reference in *theRect* and returns **self**. The frame rectangle is always reckoned in the screen coordinate system.

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*. A pointer to the screen where the Window is located is provided in the variable 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:**

getMaxSize:

– **getMaxSize:**(NXSize *)*aSize*

Returns, by reference in *aSize*, an NXSize structure that gives the maximum size to which the Window's frame can be sized by the user or by the **setFrame:...** methods. Note that this constraint doesn't apply to **sizeWindow::** or the **placeWindow:...** methods.

See also: – **setMaxSize:**, – **setMinSize:**, – **getMinSize:**

getMinSize:

– **getMinSize:**(NXSize *)*aSize*

Returns, by reference in *aSize*, an NXSize structure that gives the minimum size to which the Window's frame can be sized by the user or by the **setFrame:...** methods. Note that this constraint doesn't apply to **sizeWindow::** or the **placeWindow:...** methods.

See also: – **setMinSize:**, – **setMaxSize:**, – **getMaxSize:**

getMouseLocation:

– **getMouseLocation:**(NXPoint *)*thePoint*

Returns, by reference in *thePoint*, the current location of the mouse reckoned 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 shouldn't 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 to match the depth of the screen it's on, 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.

You never invoke this method directly; it's invoked during automatic pagination when printing (or faxing) the Window. 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 values 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)*backingType*
buttonMask:(int)*mask*
defer:(BOOL)*flag*

Initializes the Window object and returns **self**. This method is the designated initializer for the Window class.

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

You usually only create titled and resizable Windows. Menu style is used by the Menu class; miniwindows, miniworld icons, and tokens (application icons) are created for you by the Application Kit. Plain Windows lack interface accouterments and should very rarely be created and displayed.

The third argument, *backingType*, specifies how the drawing done in the Window is buffered by the object's window device:

NX_BUFFERED
NX_RETAINED
NX_NONRETAINED

The fourth argument, *mask*, specifies whether the Window's title bar will sport a close or resize button. You build the mask by joining (with the bitwise OR operator) the individual masks for the buttons:

NX_CLOSEBUTTONMASK
NX_MINIATURIZEBUTTONMASK

The fifth argument, *flag*, determines whether the Window Server will create a window device for the new object immediately. If *flag* is YES, it will defer creating the window until the Window 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 an instance of View 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 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 mechanism is currently disabled (because of a previous **disableDisplay** message), and NO if it isn't.

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 is excluded from the application's Windows menu, and NO if it isn't.

See also: – **setExcludedFromWindowsMenu:**

isFlushWindowDisabled

– (BOOL)**isFlushWindowDisabled**

Returns YES if the Window's flushing ability has been disabled; otherwise returns NO.

See also: – **disableFlushWindow**, – **reenableFlushWindow**

isKeyWindow

– (BOOL)**isKeyWindow**

Returns YES if the Window is the key window for the application, and NO if it isn't.

See also: – **isMainWindow**

isMainWindow

– (BOOL)**isMainWindow**

Returns YES if the Window is the main window for the application, and NO if it isn't.

See also: – **isKeyWindow**

isOneShot

– (BOOL)**isOneShot**

Returns YES if the window device that the Window 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 on-screen (even if it's obscured by other Windows).

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 object; typically, it's 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 (within its tier) and makes it the key window. This method can be used in action message. Returns **self**.

See also: – **orderFront:**, – **orderBack:**, – **orderOut:**, – **orderWindow:relativeTo:**

makeKeyWindow

– **makeKeyWindow**

Makes the 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 in the Window's miniwindow.

See also: – **setMiniwindowIcon:**

miniwindowImage

– (NXImage *)**miniwindowImage**

Returns the NXImage object that's displayed in the Window's miniwindow.

See also: – **setMiniwindowImage:**

miniwindowTitle

– (const char *)**miniwindowTitle**

Returns the title that's displayed in the Window's miniwindow.

See also: – **setMiniwindowTitle:**

moveTo::

– **moveTo:**(NXCoord)*x* **:**(NXCoord)*y*

Moves the Window by the lower left corner of its frame rectangle. The arguments are taken in the screen coordinate system. 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*

Moves the Window by the top left corner of its frame rectangle. The arguments are taken in the screen coordinate system. 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 empty, 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. Returns **self**.

See also: – **orderFront:**, – **orderOut:**, – **orderWindow:relativeTo:**, – **makeKeyAndOrderFront:**

orderFront:

– **orderFront:***sender*

Moves the Window to the front of its tier in the screen list. It may also change the key window and main window. Returns **self**.

See also: – **orderBack:**, – **orderOut:**, – **orderWindow:relativeTo:**, – **makeKeyAndOrderFront:**

orderFrontRegardless

– **orderFrontRegardless**

Moves the Window to the front of its tier, even if the Window's application isn't active. Normally a Window can't be moved in front of the key window unless the Window and the key window are in the same application. You should rarely need to invoke this method; it's designed to be used when applications are cooperating such that an active application (with the key window) is using another application to display data.

See also: – **orderFront:**

orderOut:

– **orderOut:***sender*

Takes the Window out of the screen list. It may also change the key window and main window. Returns **self**.

See also: – **orderFront:**, – **orderBack:**, – **orderWindow:relativeTo:**

orderWindow:relativeTo:

– **orderWindow:**(int)*place* **relativeTo:**(int)*otherWin*

Repositions the Window's window device 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 in its tier. Returns **self**.

See also: – **orderFront:**, – **orderBack:**, – **orderOut:**, – **makeKeyAndOrderFront:**

performClose:

– **performClose:***sender*

Simulates the user clicking the close button by momentarily highlighting the button and then closing the Window. If the Window's delegate or the Window itself implements **windowWillClose:**, then that message is sent with the Window as the argument (only one such message is sent; if both the delegate and the Window implement the method, only the delegate will receive the message).

If the Window doesn't have a close button, then the method calls **NXBeep()**. 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. If the Window doesn't have a miniaturize button, then this method calls **NXBeep()**. 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. You never invoke this method directly; it's automatically invoked when the Window is printed or faxed. 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 and moves the Window. *frameRect* specifies the Window's new frame rectangle in screen coordinates. The Window's frame view—but none of its other Views—is automatically redisplayed at its new size and location. Returns **self**.

See also: – **sizeWindow::**, – **moveTo::**, – **placeWindowAndDisplay:**

placeWindow:screen:

– **placeWindow:**(const NXRect *)*frameRect* **screen:**(const NXScreen *)*aScreen*

This is the same as **placeWindow:**, 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 exactly the same as **placeWindow:**. Returns **self**.

See also: – **placeWindow:**, – **placeWindowAndDisplay:**

placeWindowAndDisplay:

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

This is the same as **placeWindow:**, except the Window's Views are redisplayed before the Window is shown. 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 View's display methods. 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**

registerForDraggedTypes:count:

– **registerForDraggedTypes:**(const char *const *)*pbTypes* **count:**(int)*count*

Registers the Pasteboard types that the Window will accept in an image-dragging session. *pbTypes* is a pointer to an array of the types; *count* is the number of elements in the array. Returns **self**.

Keep in mind that the values in the first argument are Pasteboard types, *not* file extensions (you can't register for specific file extensions). For example, the following registers a Window as accepting files:

```
const char *fileType[] = {NXFilenamePboardType};
[aWindow registerForDraggedTypes:fileType count:1];
```

Note: Registering a Window for dragged types automatically makes it a candidate destination object during a dragging session. As such, it must implement some or all of the NXDraggingDestination protocol methods. As a convenience, Window provides default implementations of these methods (in general, the dragging destination methods are forwarded to the Window's delegate). See the NXDraggingDestination protocol description for details.

See also: – **unregisterDraggedTypes**

removeCursorRect:cursor:forView:

– **removeCursorRect:**(const NXRect *)*aRect*
cursor:*anObj*
forView:*aView*

Removes a cursor rectangle from the Window. You never invoke this method; it's used by View's **removeCursorRect:cursor:** method. To remove a cursor rectangle, use the View method.

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.

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 invoke **invalidateCursorRectsForView:**, rather than this method, to fix invalid cursor rectangles.

See also: – **invalidateCursorRectsForView:**, – **resetCursorRects** (View)

resignKeyWindow

– **resignKeyWindow**

You never invoke this method; it's invoked automatically when the Window resigns key window status. The method sends **resignKeyWindow** to the Window's first responder, and **sends windowDidResignKey:** to the Window's delegate (if the respective objects can respond). Returns **self**.

See also: – **becomeKeyWindow**

resignMainWindow

– **resignMainWindow**

You never invoke this method; it's invoked automatically when the Window resigns main window status. The method sends **windowDidResignMain:** to the Window's delegate (if the delegate can respond). Returns **self**.

See also: – **becomeMainWindow**

resizeFlags

– (int)**resizeFlags**

Valid only while the Window is being resized, this method returns the *flags* field of the event record for the mouse-down event that initiated the resizing session. The integer encodes, as a mask, information such as which of the modifier keys was held down when the event occurred. The flags are listed in **dpsclient/event.h**. Because of its limited validity, this method should only be invoked from within an implementation of the delegate methods **windowWillResize:toSize:** or **windowDidResize:**.

rightMouseDown:

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

Responds to uncaught right mouse-down events by forwarding this message to 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 value returned by the Application object.

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

saveFrameToString:

– (void)**saveFrameToString:**(char *)*string*

Saves the Window's frame rectangle data as a NULL-terminated ASCII string to the buffer pointed to by *string*. The string can be stored as you see fit and used later to set the dimensions of a Window through the **setFrameFromString:** method. You should use the constant NX_MAXFRAMESTRINGLENGTH to allocate the buffer.

See also: – **setFrameFromString:**, – **saveFrameUsingName:**

saveFrameUsingName:

– (void)**saveFrameUsingName:**(const char *)*name*

Saves the Window's frame rectangle as a system default. With the companion method **setFrameUsingName:**, you can save and reset a Window's frame over various launchings of an application. The default is owned by the application, filed under the name

"Window Frame *name*"

See also: – **setFrameUsingName:**, – **saveFrameToString:**

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*

Invoked when the user releases the Window, having moved all or part of it to a different screen. This method sends the delegate a **windowDidChangeScreen:** message (if the delegate can respond) and returns **self**.

If the Window has a dynamic depth limit, this method will make sure that the depth limit matches the new device. If the Window is on more than one screen, its depth limit will be adjusted to match the deepest screen it's on.

See also: – **bestScreen**

sendEvent:

– **sendEvent:**(NXEvent *)*theEvent*

Dispatches mouse and keyboard events sent to the Window by the Application object; you never invoke this method directly.

setAvoidsActivation:

– **setAvoidsActivation:**(BOOL)*flag*

Establishes whether the Window's application will become the active application when the user clicks in the Window's content area. If *flag* is YES, the application won't become active; if *flag* is NO, it will. The default is NO. Note that clicking on the title bar will always activate the Window's application.

See also: – **avoidsActivation**

setBackgroundColor:

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

Sets the color that fills the Window's content area when the Window is displayed on a color screen. Returns **self**.

See also: – **backgroundColor**

setBackgroundGray:

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

Sets the shade of gray that fills the Window's content area when the Window is displayed on a monochrome screen. *value* should lie in the range 0.0 (black) to 1.0 (white). Returns **self**.

See also: – **backgroundGray**, – **setBackgroundColor:**

setBackingType:

– **setBackingType:**(int)*backing*

Sets the type of backing used by the Window's window device and returns **self**. This method can only be used to switch a buffered Window to nonretained or vice versa; you can't change the backing type of a nonretained Window (a PostScript error is generated if you attempt to do so).

See also: – **backingType**

setContentView:

– **setContentView:***aView*

Makes *aView* the Window's content view; the previous content view is removed from the Window's view hierarchy and returned by this method. *aView* is resized to fit precisely within the content area of the Window. You can transform the content view's coordinate system, but you can't alter its size or location directly.

See also: – **contentView**

setDelegate:

– **setDelegate:***anObject*

Makes *anObject* the Window's delegate, and returns **self**. A Window's 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 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:

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; the original event mask is returned. 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_TIMERMASK
NX_JOURNALEVENTMASK
NX_NULLEVENTMASK

Miniwindows and application icons have the same default event mask as other Windows, except that 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**

setFrameAutosaveName:

– (BOOL)**setFrameAutosaveName:**(const char *)*name*

Sets the name that's used to automatically save the Window's frame rectangle in the defaults system. If *name* isn't NULL, the Window's frame is saved as a default (as described in **saveFrameUsingName:**) each time the frame changes. Passing NULL as an argument turns off this automation. A Window can have only one frame autosave name at a time; if the Window already has an autosave name, the old one is replaced. If *name* is already being used as an autosave name by a Window in this application, the name isn't set and this method returns NO; otherwise returns YES.

See also: **– setFrameUsingName:**, **+ removeFrameUsingName:**, **– saveFrameToString:**, **– setFrameFromString:**

setFrameFromString:

– (void)**setFrameFromString:**(const char *)*data*

Sets the Window's frame rectangle by reading the frame rectangle data stored in *data*. The data should have been previously stored through the **saveFrameToString:** method. The frame is constrained according to the Window's minimum and maximum size settings. This method causes a **windowWillResize:toSize:** message to be sent to the delegate.

See also: **– saveFrameToString:**

setFrameUsingName:

– (BOOL)**setFrameUsingName:**(const char *)*name*

Sets the Window's frame rectangle by reading, from the defaults system, the rectangle data stored in *name*. The frame is constrained according to the Window's minimum and maximum size settings. This method causes a **windowWillResize:toSize:** message to be sent to the delegate.

If *name* doesn't exist, the frame isn't set and this method returns NO; otherwise returns YES.

See also: – **setFrameAutosaveName:**, + **removeFrameUsingName:**, – **saveFrameToString:**, – **setFrameFromString:**

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:**

setMaxSize:

– **setMaxSize:**(const NXSize *)*aSize*

Sets the maximum size to which the user can resize the Window's frame rectangle. Note that this constraint isn't enforced when the Window is resized through the **sizeWindow::** or **placeWindow...** methods. Returns **self**.

See also: – **getMaxSize:**, – **setMinSize:**, – **getMinSize:**

setMinSize:

– **setMinSize:**(const NXSize *)*aSize*

Sets the minimum size to which the user can resize the Window's frame rectangle. Note that this constraint isn't enforced when the Window is resized through the **sizeWindow::** or **placeWindow...** methods. Returns **self**.

See also: – **getMaxSize:**, – **setMinSize:**, – **getMinSize:**

setMiniwindowIcon:

– **setMiniwindowIcon:**(const char *)*name*

Sets the image that's displayed by the Window's miniwindow. The named icon is searched for using NXImage's **findImageNamed:** class method. This method is guaranteed to work only if it's invoked from within an implementation of the **windowWillMiniaturize:toMiniwindow:** delegate method, or if the miniwindow is currently visible.

See also: – **setMiniwindowImage:**, – **miniwindowIcon**

setMiniwindowImage:

– **setMiniwindowImage:***image*

Sets the image that's displayed by the Window's miniwindow. The argument should be an NXImage object. This method is guaranteed to work only if it's invoked from within an implementation of the **windowWillMiniaturize:toMiniwindow:** delegate method, or if the miniwindow is currently visible.

See also: – **setMiniwindowIcon:**, – **miniwindowImage**

setMiniwindowTitle:

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

Sets the title of the Window's miniwindow. Normally, the miniwindow's title is taken, often abbreviated, from that of the Window. This method is guaranteed to work only if it's invoked from within an implementation of the **windowWillMiniaturize:toMiniwindow:** delegate method, or if the miniwindow is currently visible. In the latter case, the miniwindow's title is automatically redisplayed. Note that setting the Window's title (through **setTitle:** or **setTitleAsFilename:**) will automatically reset the miniwindow's title to that of the Window.

See also: – **miniwindowTitle:**, – **setTitle**, – **setTitleAsFilename:**

setOneShot:

– **setOneShot:**(BOOL)*flag*

Sets whether the window device 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*

Sets the string that appears in the Window's title bar (if it has one). This also sets the title of the Window's miniwindow. The new title is automatically displayed. Returns **self**.

See also: – **title**, – **setTitleAsFilename:**, – **setMiniwindowTitle:**

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

This method also sets the title of the Window's miniwindow.

Returns **self**.

See also: – **title**, – **setTitle:**, – **setMiniwindowTitle:**

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. The arguments are:

- *aRect* is a pointer to the tracking rectangle specified in the Window's coordinate system.
- *insideFlag* is YES if the cursor starts off inside the rectangle, otherwise it's NO.
- *anObject* is the object, usually a View or an NXCursor, that will handle the mouse-entered and mouse-exited events that are generated for the rectangle.
- *trackNum* is a tag you assign to identify the rectangle.
- If *leftDown* is YES, mouse-entered and mouse-exited events are generated only while the left mouse button is down.
- If *rightDown* is YES, mouse 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*

This does for faxing what **smartPrintPSCode** does for printing. 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:**

smartPrintPSCode:

– **smartPrintPSCode:***sender*

Prints the Window (all the Views in its view hierarchy including the frame view) on a single sheet of paper. 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—they don't 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:**

spoolFile:

– **spoolFile:**(const char *)*filename*

Spools the 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 the following values, indicating the Window's style:

NX_PLAINSTYLE
NX_TITLEDSTYLE
NX_RESIZEBARSTYLE
NX_MENUSTYLE
NX_MINIWINDOWSTYLE
NX_MINIWORLDSTYLE
NX_TOKENSTYLE

A Window's style is set when the object is initialized. Once set, it can't be changed.

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

title

– (const char *)**title**

Returns the string that appears in the title bar of the window.

See also: – **setTitle:**, – **setTitleAsFilename:**

tryToPerform:with:

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

Gives the Window's delegate a chance to respond to the action message before passing the message up the responder chain. If a receiver for *anAction* is found, this method returns YES. Otherwise, it returns NO.

See also: – **tryToPerform:with:** (Responder)

unregisterDraggedTypes

– **unregisterDraggedTypes**

Unregisters the Window as a possible recipient of dragged-images.

See also: – **registerForDraggedTypes:count:**

update

– **update**

The default implementation of this method does nothing more than send a **windowDidUpdate:** message to the Window's delegate (if the delegate can respond) and return **self**. A subclass can reimplement this method to perform specialized operations, but should send an **update** message to **super** just before returning. For example, the Menu class implements this method to disable and enable menu commands as appropriate.

A Window is automatically sent an **update** message before it's ordered into the screen list. If the Application object has received a **setAutoupdate:YES** message, each visible Window in the application is sent an **update** message after every event in the main event loop. A Panel object that isn't visible is sent an **update** message as part of its implementation of the **commandKey:** method.

You can manually cause an update message to be sent to all visible Windows through Application's **updateWindows** method.

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. You should 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*

Invoked when a portion of the Window that was previously obscured is uncovered. This method is only invoked if the Window's backing is nonretained. The Views in the uncovered portion of the Window are redisplayed, and a **windowDidExpose:** message is sent to the delegate. Returns **self**.

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

windowMoved:

– **windowMoved:**(NXEvent *)*theEvent*

Invoked when the user moves the Window. A **windowDidMove:** message is sent to the delegate. Returns **self**.

See also: – **dragFrom::eventNum:**, – **setDelegate:**

windowNum

– (int)**windowNum**

Returns the window number of the Window's window device. Each window device in an application is given a unique window number—note that this isn't the same as the global window number assigned by the Window Server.

If the Window doesn't have a window device, the return value will be equal to or less than 0.

See also: – **initContent:style:backing:buttonMask:defer:**, – **setOneShot:**, **NXConvertWinNumToGlobal()**

worksWhenModal

– (BOOL)**worksWhenModal**

Returns YES if the Window is able to receive keyboard and mouse events 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*

Invoked when the *sender* Window becomes the key window.

windowDidBecomeMain:

– **windowDidBecomeMain:***sender*

Invoked when the *sender* Window becomes the main window.

windowDidChangeScreen:

– **windowDidChangeScreen:***sender*

Invoked when the user finishes moving the *sender* Window to a different screen.

See also: – **screenChanged:**

windowDidDeminiaturize:

– **windowDidDeminiaturize:***sender*

Invoked when 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*

Invoked when a previously obscured portion of the nonretained *sender* Window is uncovered.

See also: – **windowExposed:**

windowDidMiniaturize:

– **windowDidMiniaturize:***sender*

Invoked after the *sender* Window has been miniaturized (whether by the user or through the **performMiniaturize:** or **miniaturize:** method).

See also: – **windowWillMiniaturize:toMiniwindow:**, – **windowDidDeminiaturize:**

windowDidMove:

– **windowDidMove:***sender*

Invoked when the user finishes moving the *sender* Window.

See also: – **windowMoved:**

windowDidResignKey:

– **windowDidResignKey:***sender*

Invoked when the *sender* Window resigns its status as key window.

windowDidResignMain:

– **windowDidResignMain:***sender*

Invoked when the *sender* Window resigns its status as main window.

windowDidResize:

– **windowDidResize:***sender*

Invoked when the user finishes resizing the *sender* Window.

See also: – **windowWillResize:toSize:**, – **getFrame:**

windowDidUpdate:

– **windowDidUpdate:***sender*

Invoked when the *sender* Window receives an **update** message.

See also: – **update**

windowWillClose:

– **windowWillClose:***sender*

Invoked just before the user (or the **performClose:** method) closes the *sender* Window. If this method returns **nil**, the Window isn't closed.

windowWillMiniaturize:toMiniwindow:

– **windowWillMiniaturize:***sender* **toMiniwindow:***miniwindow*

Invoked before the sender Window is miniaturized (whether by the user or through the **performMiniaturize:** or **miniaturize:** method). The return value is ignored.

See also: – **windowDidMiniaturize:**, – **miniaturize:**

windowWillMove:

– **windowWillMove:***sender*

Invoked when the user clicks on the title bar of the *sender* Window. Note that this method isn't sent when the user drags the Window by clicking in a Window-dragging View (as defined by the **dragFrom::eventNum:** method). The return value is ignored.

windowWillResize:toSize:

– **windowWillResize:***sender* **toSize:**(NXSize *)*frameSize*

Invoked when the *sender* Window is being resized (whether by the user or through one of the **setFrame...** methods). The second argument, *frameSize*, contains the size (in screen coordinates) that the Window will be resized to. To reset the size, simply set *frameSize* directly from this method. The Window's minimum and maximum size constraints will already have been applied when this method is invoked.

If the user is resizing the Window, the delegate is sent a series of **windowWillResize:toSize:** messages as the Window's outline is dragged. The Window's outline will be displayed at the constrained size as set by this method.

See also: – **windowDidResize:**

windowWillReturnFieldEditor:toObject:

– **windowWillReturnFieldEditor:***sender* **toObject:***client*

Invoked when the *sender* Window's field editor is requested by *client*. 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:**

Protocols

NXChangeSpelling

Adopted By: Text

Declared In: appkit/NXSpellChecker.h

Protocol Description

An object in the responder chain that can correct a misspelled word implements this protocol.

See also: NXSpellChecker class

Instance Methods

changeSpelling:

– **changeSpelling:***sender*

Replaces with a corrected version the selected word in the object whose text is being checked. This message is sent by the NXSpellChecker instance to the object whose text is being checked. To get the corrected spelling, the receive asks the sender for the string value of its selected cell (visible to the user as the text field in the Spelling Panel). The method replaces the selected portion of the text stream with the string that it gets from the NXSpellChecker.

Returns **self** when the replacement is successful, and **nil** otherwise (for example, if the selected text is in an object that is not editable).

NXColorPickingCustom

Adopted By: no NeXTSTEP classes

Declared In: appkit/colorPicking.h

Protocol Description

Together with the NXColorPickingDefault protocol, NXColorPickingCustom provides a way to add color pickers—custom user interfaces for color selection—to an application's NXColorPanel. The NXColorPickingDefault protocol provides basic behavior for a color picker. The NXColorPicker class adopts the NXColorPickingDefault protocol. The easiest way to implement a color picker is to create a subclass of NXColorPicker and use it to implement the NXColorPickingCustom protocol.

See also: NXColorPickingDefault, NXColorPicker (class)

Method Types

View management	– provideNewView:
Mode	– supportsMode: – currentMode
Setting color	– setColor:

Instance Methods

currentMode

– (int)**currentMode**

Returns the current mode of the color picker. The returned value should be unique to your color picker. Unique values for the standard color pickers are defined in NXColorPanel.h; they are:

NX_GRAYMODE	0
NX_RGBMODE	1
NX_CMYKMODE	2
NX_HSBMODE	3
NX_CUSTOMPALETTEMODE	4
NX_COLORLISTMODE	5
NX_WHEELMODE	6

If your color picker includes submodes, you should define a unique integer for each submode. As an example, the slider picker has four values defined in the above list (NX_GRAYMODE, NX_RGBMODE, NX_CMYKMODE, and NX_HSBMODE), one for each of its submodes.

provideNewView:

– **provideNewView:**(BOOL)*firstRequest*

This method returns a view containing the user interface for the color picker. This message is sent to your color picker whenever the color panel attempts to display it. This may be when the panel is first presented, when the user switches pickers, or when the picker is switched through API. *firstRequest* is YES only when this method is first invoked for your color picker—you may use this opportunity to lazily load nib files, initialize the view and perform any other custom initialization required for your picker. The View returned by this method should be set to automatically resize both its width and height.

setColor:

– **setColor:**(NXColor)*newColor*

Sets the color of the color picker. This method is invoked on the current color picker each time NXColorPanel's **setColor:** method is invoked. If *newColor* is actually different from the color picker's color (as it would be if, for example, the user dragged a color into NXColorPanel's color well) this method could be used to update the color picker's color to reflect the change.

supportsMode:

– (BOOL)**supportsMode:**(int)*mode*

Returns YES if the NXColorPicking protocol implementor supports the picking mode represented by *mode*. This method is invoked when the NXColorPanel's is first initialized—it is used to attempt to restore the user's previously selected mode. It is also invoked by NXColorPanel's **setMode:** to find the color picker that supports a particular mode.

NXColorPickingDefault

Adopted By: NXColorPicker

Declared In: appkit/colorPicking.h

Protocol Description

The NXColorPickingDefault protocol, together with the NXColorPickingCusom protocol, provides API for adding color pickers—custom user interfaces for color selection—to an application's NXColorPanel. The NXColorPickingDefault protocol provides basic behavior for a color picker. The NXColorPickingCustom protocol provides implementation-specific behavior.

The NXColorPicker class implements the NXColorPickingDefault protocol. The simplest way to implement your own color picker is to create a subclass of NXColorPicker and implement the NXColorPickingCustom protocol in that subclass. You can also implement custom color selection by creating a subclass of another class such as View, and using that subclass to implement the methods in this protocol and in NXColorPickingCustom.

Color Picker Bundles

A class that implements the NXColorPickingDefault and NXColorPickingCustom protocols needs to be compiled and linked in an application's object file. However, your application need not explicitly create an instance of this class. Instead, your application's file package should include a directory named ColorPickers; within this directory you should place a directory "*MyPickerClass*.bundle" for each custom color picker your application implements. This bundle should contain all resources required for your color picker: nib files, tiff files, and so on.

NXColorPanel will allocate and initialize an instance of each class for which a bundle is found in the "ColorPickers" directory.

Color Picker Buttons

NXColorPanel lets the user select a color picker from a Matrix of ButtonCells. The order in which a color picker's ButtonCell appears in this Matrix is determined by the return value of its **insertionOrder** method.

See also: NXColorPickingCustom, NXColorPicker (class), NXColorPanel (class)

Method Types

Initialization	– initFromPickerMask:withColorPanel:
Button images	– provideNewButtonImage – insertNewButtonImage:in:
View management	– viewSizeChanged:
Alpha control check	– alphaControlAddedOrRemoved:
Order of button appearance	– insertionOrder
Using color lists	– attachColorList: – detachColorList: – updateColorList:
Mode	– setMode:

Instance Methods

alphaControlAddedOrRemoved:

– **alphaControlAddedOrRemoved:***sender*

Invoked automatically when the NXColorPanel's opacity slider is added or removed; you never invoke this method directly.

You can determine whether the slider is being added or removed by sending the **doesShowAlpha** message to *sender*. A return of YES means that the *sender* is displaying the opacity (alpha) slider; NO means it isn't.

attachColorList:

– **attachColorList:***colorList*

Tells the color picker to attach the given *colorList* (an NXColorList object). You never invoke this method; it's invoked automatically by the NXColorPanel when its **attachColorList:** method is invoked. Use this method if you implement a custom color picker that manages NXColorLists. If the color picker isn't displaying *colorList*, it should be added to the picker. This method ordinarily needs not do anything, since NXColorPanel's list mode manages NXColorLists. Returns **self**.

See also: – **attachColorList:** (NXColorPanel), NXColorList class

detachColorList:

– **detachColorList:***colorList*

Tells the color picker to detach the given *colorList* (an NXColorList object). You never invoke this method; it's invoked automatically by the NXColorPanel when its **detachColorList:** method is invoked. Use this method if you implement a custom color picker that manages NXColorLists. If the color picker is displaying *colorList*, it should be removed from the picker. This method ordinarily needs not do anything, since NXColorPanel's list mode manages NXColorLists. Returns **self**.

initFromPickerMask:withColorPanel:

– **initFromPickerMask:**(int)*mask* **withColorPanel:***appPanel*

Notifies the color picker of the color panel's mask and initializes the color picker. This method is sent by the NXColorPanel to all implementors of the color picking protocols when the application's color panel is first initialized. In order for your color picker to receive this message, it must have a bundle in your application's "ColorPickers" directory (described in "Color Picker Bundles" in the Protocol Description).

mask is determined by the argument to the NXColorPanel method **setPickerMask:**. If no mask has been set, *mask* is NX_ALLMODESMASK. If your color picker supports any additional modes, you should invoke the **setPickerMask:** method when your application intializes to notify the NXColorPanel class.

This method should examine the mask and determine whether it supports any of the modes included there. You may also check the value in *mask* to enable or disable any subpickers or optional controls implemented by your color picker. Your color picker may also retain *appPanel* in an instance variable for future communication with the color panel.

This method is provided to initialize your color picker; however, much of a color picker's initialization may be done lazily through the **provideNewView:** method. If your color picker responds to any of the modes represented in *mask*, it should perform its initialization and return **self**. Color pickers that do so will have their buttons inserted in the color panel and will continue to receive messages from the panel as the user manipulates it. If the color picker doesn't respond to any of the modes represented in *mask*, it should do nothing and return **nil**.

See also: + **setPickerMask:** (NXColorPanel class), – **provideNewView:**

insertNewButtonImage:in:

– **insertNewButtonImage:***newImage* **in:***newButtonCell*

Sets *newImage* as *newButtonCell*'s image. *buttonCell* is the ButtonCell object that displays the color picker's representation in the NXColorPanel's picker Matrix—the control that lets the user choose the picker to use. Implement this method to perform application-specific manipulation of the image before it is inserted and displayed by the button cell.

See also: – **insertionOrder**, – **provideNewButtonImage:**

insertionOrder

– (float)**insertionOrder**

Returns a float value representing the insertion order of the receiver's ButtonCell in the NXColorPanel's picker Matrix—the control that lets the user choose the picker to use. Values representing the insertion order of the standard color pickers are defined in the header file **appkit/NXColorPanel.h**. The standard color pickers used by NXColorPanel have symbolic constants (defined in NXColorPanel.h) that determine their insertion order:

Constant	Value
NX_WHEEL_INSERTION	(0.50)
NX_SLIDERS_INSERTION	(0.51)
NX_CUSTOMPALETTE_INSERTION	(0.52)
NX_LIST_INSERTION	(0.53)

Implement this method to place your color picker's ButtonCell in the desired position relative to these default color pickers. For example, to place the ButtonCell for your color picker between those for wheel and slider, implement a version of this method that returns 0.505.

See also: – **insertNewButtonImage:in:**, – **provideNewButtonImage**

provideNewButtonImage

– **provideNewButtonImage**

This method should return an NXImage to represent the color picker in the NXColorPanel's picker Matrix: the Matrix of ButtonCells from which the user selects a color picker.

See also: – **insertNewButtonImage:in:**, – **insertionOrder**

setMode:

– **setMode:**(int)*mode*

Sets the mode of the color picker. This method is invoked by NXColorPanel's **setMode:** method to ensure that all color pickers reflect the current mode—for example, upon color picker initialization to ensure that all color pickers are restored to the mode the user left them in the last time an NXColorPanel was used.

Most color picker's have only one mode, and thus don't need to do any work in this method. An example of a color picker that uses this method is the slider picker, which can choose from one of several submodes depending on the value of *mode*. Returns **self**.

updateColorList:

– **updateColorList:***colorList*

Tells the color picker when a color list has been updated. This method is invoked when NXColorPanel's **updateCustomColorList:** method is invoked. If *colorList* is visible in the color picker, it should be updated; if colorList is **nil**, all color lists currently visible in the color picker should be updated.

viewSizeChanged:

– **viewSizeChanged:***sender*

Tells the color picker when the NXColorPanel's view size changes. *sender* is the sending NXColorPanel. Use this method to perform special preparation when resizing the color picker's view. Since this method is invoked only as appropriate, it is better to implement this method than to override the method **superviewSizeChanged:** for the View in which the color picker's user interface is contained.

See also: – **provideNewView:** (NXColorPickingCustom)

NXDraggingDestination

Adopted By: no NeXTSTEP classes

Declared In: appkit/drag.h

Protocol Description

The NXDraggingDestination protocol declares methods that the destination (or recipient) of a dragged image must implement. The destination automatically receives NXDraggingDestination messages as an image enters, moves around inside, and then exits or is released within the destination's boundaries.

Note: In the text here and in the other dragging protocol descriptions, the term *dragging session* is the entire process during which an image is selected, dragged, released, and is absorbed or rejected by the destination. A *dragging operation* is the action that the destination takes in absorbing the image when it's released. The *dragging source* is the object that "owns" the image that's being dragged. It's specified as an argument to the **dragImage:...** message, sent to a Window or View, that instigated the dragging session.

The Dragged Image

The image that's dragged in an image-dragging session is an NXImage object that represents data that's put on the pasteboard. Although a dragging destination can access the NXImage (through a method described in the NXDraggingInfo protocol), its primary concern is with the pasteboard data that the NXImage represents—the dragging operation that a destination ultimately performs is on the pasteboard data, not on the image itself.

Valid Destinations

Dragging is a visual phenomenon. To be an image-dragging destination, an object must represent a portion of screen real estate; thus, only Windows and Views can be destinations. Furthermore, you must announce the destination-candidacy of a Window or View by sending it a **registerForDraggedTypes:count:** message. This method, defined in both classes, registers the pasteboard types that the object will accept. During a dragging session, a candidate destination will only receive NXDraggingDestination messages if the pasteboard types for which it is registered matches a type that's represented by the image that's being dragged.

Although NXDraggingDestination is declared as a protocol, the Views and Window subclasses that you create to adopt the protocol need only implement those methods that are pertinent. (The View and Window classes provide private implementations for all of the methods.) In addition, a Window or its delegate may implement these methods; the delegate's implementation takes precedent.

The Sender of Destination Messages

Each of the NXDraggingDestination methods sports a single argument, *sender*, the object that invoked the method. Within its implementations of the NXDraggingDestination methods, the destination can send NXDraggingInfo messages to *sender* to get more information on the current dragging session.

The Order of Destination Messages

The six NXDraggingDestination methods are invoked in a distinct order:

- As the image is dragged into the destination's boundaries, the destination is sent a **draggingEntered:** message
- While the image remains within the destination, a series of **draggingUpdated:** messages are sent.
- If the image is dragged out of the destination, **draggingExited:** is sent and the sequence of NXDraggingDestination messages stops. If it re-enters, the sequence begins again (with a new **draggingEntered:** message).
- When the image is released, it either slides back to its source (and breaks the sequence) or a **prepareForDragOperation:** message is sent to the destination, depending on the value that was returned by the most recent invocation of **draggingEntered:** or **draggingUpdated:**.
- If the **prepareForDragOperation:** message returned YES, a **performDragOperation:** message is sent.
- Finally, if **performDragOperation:** returned YES, **concludeDragOperation:** is sent.

Method Types

Before the image is released	– draggingEntered: – draggingUpdated: – draggingExited:
After the image is released	– prepareForDragOperation: – performDragOperation: – concludeDragOperation:

Instance Methods

concludeDragOperation:

– **concludeDragOperation:**(id <NXDraggingInfo>)*sender*

Invoked when the dragging operation is complete (but only if the previous **perfromDragOperation:** returned YES). The destination implements this method to perform any tidying up that it needs to do. This is the last message that's sent from *sender* to the destination during a dragging session. The return value is ignored.

draggingEntered:

– (NXDragOperation)**draggingEntered:**(id <NXDraggingInfo>)*sender*

Invoked when the dragged image enters the destination. Specifically, the message is sent when the hot spot on the cursor that's dragging the image enters any portion of the destination's bounds rectangle (if it's a View) or its frame rectangle (if it's a Window).

The method must return a single NXDragOperation value that indicates which dragging operation the destination will perform when the image is released. It should be one of the operations specified in the value returned by *sender*'s **draggingSourceOperationMask** method. If none of the operations are appropriate, this method should return NX_DragOperationNone (this is the default response if the method isn't implemented by the destination). The dragging operation constants are listed in the "Types and Constants" section of this chapter

See also: – **draggingUpdated:**, – **draggingExited:**, – **prepareForDragOperation:**

draggingExited:

– **draggingExited:**(id <NXDraggingInfo>)*sender*

Invoked when the dragged image exits the destination (following, inversely, the geometric specification given in the description of **draggingEntered:**). The return value is ignored.

draggingUpdated:

– (NXDragOperation)**draggingUpdated:**(id <NXDraggingInfo>)*sender*

Invoked periodically as the image is poised within the destination. The messages continue until the image is either released or exits. The return value follows the same rules as that for the **draggingEntered:** method. The default return value (if this method isn't implemented by the destination) is the value returned by the previous **draggingEntered:** message.

Only one destination at at time receives a sequence of **draggingUpdated:** messages. For example, if the cursor is within the bounds of two overlapping Views that are both valid destinations, the uppermost View receives these messages until the image is either released or exits.

See also: – **draggingExited:**, – **prepareForDragOperation:**

performDragOperation:

– (BOOL)**performDragOperation:**(id <NXDraggingInfo>)*sender*

Invoked after the released image has been removed from the screen (but only if the previous **prepareForDragOperation:** message returned YES). The destination should implement this method to do the real work of importing the data represented by the image. If the destination accepts the data, it returns YES, otherwise it returns NO. The default (if the destination doesn't implement the method) is to return NO.

See also: – **concludeDragOperation:**

prepareForDragOperation:

– (BOOL)**prepareForDragOperation:**(id <NXDraggingInfo>)*sender*

Invoked when the image is released (but only if the most recent **draggingEntered:** or **draggingUpdated:** message returned an acceptable drag-operation value). The method returns YES if it will perform the drag operation and NO if not.

See also: – **performDragOperation:**

NXDraggingInfo

Adopted By: no NeXTSTEP classes

Declared In: appkit/drag.h

Protocol Description

The NXDraggingInfo protocol declares methods that supply information about a *dragging session* (see the NXDraggingDestination protocol for definitions of dragging terms). The NXDraggingInfo methods are designed to be invoked from within an object's implementation of the NXDraggingDestination protocol methods. An NXDraggingInfo-conforming object is passed as the argument to each of the methods defined by NXDraggingDestination; it is to this object that the NXDraggingInfo messages should be sent. The Application Kit supplies an NXDraggingInfo object automatically such that you never need to create a class that implements this protocol.

Method Types

Dragging-session information	– isDraggingSourceLocal – draggingSource – draggingSourceOperationMask – draggingDestinationWindow – draggingPasteboard – draggingSequenceNumber – draggingLocation
Image information	– draggedImage – draggedImageCopy – draggedImageLocation
Sliding the image	– slideDraggedImageTo:

Instance Methods

draggedImage

– (NXImage *)**draggedImage**

Returns the NXImage object that's being dragged. You shouldn't invoke this method after the user has released the image, nor should you free the object that this method returns.

See also: – **draggedImageCopy**, – **draggedImageLocation**

draggedImageCopy

– (NXImage *)**draggedImageCopy**

Returns a copy of the NXImage object that's being dragged. You should only invoke this method after the user has released the image, typically from within the destination's implemenation of **performDragOperation:**. You must free the NXImage when you're done with it.

See also: – **draggedImage**, – **draggedImageLocation**

draggedImageLocation

– (NXPoint)**draggedImageLocation**

Returns the current location of the dragged image's origin. The image moves in lockstep with the cursor (the position of which is given by **draggingLocation**) but may be positioned at some offset. The point that's returned is reckoned in the base coordinate system of the destination object's Window.

See also: – **draggedImage**, – **draggingLocation**

draggingDestinationWindow

– **draggingDestinationWindow**

Returns the destination object's Window.

draggingLocation

– (NXPoint)**draggingLocation**

Returns the current location of the cursor's hot spot, reckoned in the base coordinate system of the destination object's Window.

See also: – **draggedImageLocation**

draggingPasteboard

– (Pasteboard *)**draggingPasteboard**

Returns the Pasteboard object that holds the data that the dragged image represents.

draggingSequenceNumber

– (int)**draggingSequenceNumber**

Returns an integer that uniquely identifies the current dragging session.

draggingSource

– **draggingSource**

Returns the source, or "owner," of the dragged image. However, this method returns **nil** if the source isn't in the same application as the destination.

See also: – **isDraggingSourceLocal**

draggingSourceOperationMask

– (NXDragOperation)**draggingSourceOperationMask**

Returns the dragging operation mask declared by the dragging source (through its **draggingSourceOperationMaskForLocal:** method). The elements in the mask will be one or more of the following:

NX_DragOperationCopy
NX_DragOperationLink
NX_DragOperationGeneric
NX_DragOperationPrivate

If the user is holding down a modifier key during the drag, the NXDragOperation value that corresponds to the key (as shown in the table below) is AND'ed with the source's mask.

Modifier Key	NXDragOperation Value
Control	NX_DragOperationLink
Alternate	NX_DragOperationCopy
Command	NX_DragOperationGeneric

isDraggingSourceLocal

– (BOOL)**isDraggingSourceLocal**

Returns YES if the source and destination objects are in the same application. Otherwise returns NO.

slideDraggedImageTo:

– **slideDraggedImageTo:**(NXPoint *)*screenPoint*

Causes the image to move to the given location in the screen coordinate system. This method should only be invoked from within the destination's implementation of **prepareForDragOperation:**—in other words, after the user has released the image but before it's removed from the screen.

NXDraggingSource

(informal protocol)

Category Of: Object

Declared In: appkit/drag.h

Category Description

The NXDraggingSource category declares methods that can (or must) be implemented by the source object in a dragging session (see the NXDraggingDestination protocol for definitions of dragging terms). This *dragging source* is specified as an argument to the **dragImage:...** message, sent to a Window or View, that instigated the dragging session.

Warning: See the documentation of the **dragImage:...** method in either Window or View for a warning regarding the mouse-down event that initiates a dragging session.

Of the methods declared below, only **draggingSourceOperationMaskForLocal:** *must* be implemented. The other two methods are invoked only if the dragging source implements them. All three methods are invoked automatically during a dragging session—you never send an NXDraggingSource message directly to an object.

Method Types

Querying the source	– draggingSourceOperationMaskForLocal:
Informing the source	– draggedImage:beganAt: – draggedImage:endedAt:deposited:

Instance Methods

draggedImage:beganAt:

– **draggedImage:**(NXImage *)*image* **beganAt:**(NXPoint *)*screenPoint*

Invoked when the dragged image, given by *image*, is displayed but before it starts following the mouse. The origin of the image is given by *screenPoint*, reckoned in screen coordinates. This method isn't invoked if the dragging source doesn't provide an implementation for it. The return value is ignored.

draggedImage:endedAt:deposited:

– **draggedImage:**(NXImage *)*image*
endedAt:(NXPoint *)*screenPoint*
deposited:(BOOL)*didDeposit*

Invoked after the dragged image has been released and the dragging destination has been given a chance to operate on the data it represents. The image is given by *image*, the location of the image's origin when it was released reckoned in screen coordinates, is given by *screenPoint*. This method isn't invoked if the dragging source doesn't provide an implementation for it. The final argument, *didDeposit*, indicates whether the destination accepted the image. The return value is ignored.

draggingSourceOperationMaskForLocal:

– (NXDragOperation)**draggingSourceOperationMaskForLocal:**(BOOL)*isLocal*

This is the only NXDraggingSource method that must be implemented by the source object. It should return an NXDragOperation mask, built by OR'ing the applicable constants listed below, that represents the operations that can be performed on the dragged image's data. The *isLocal* flag indicates whether the candidate destination object (the Window or View over which the dragged image is currently poised) is in the same application as the source.

NXDragOperation	Meaning
NX_DragOperationNone	No operation is possible.
NX_DragOperationCopy	The data represented by the image can be copied.
NX_DragOperationLink	The data can be shared.
NX_DragOperationGeneric	The operation can be defined by the destination.
NX_DragOperationPrivate	Private source/destination negotiation.
NX_DragOperationAll	Combines all the above.

NXIgnoreMisspelledWords

Adopted By: Text

Declared In: appkit/NXSpellChecker.h

Protocol Description

This protocol permits an NXSpellChecker object to match the requests it receives with the documents they belong to. The need arises because the list of ignored words is typically attached to a *document*, but that's not the way an NXSpellChecker receives its requests.

When the NXSpellChecker receives a request to check spelling, the request is made on behalf of an object that contains a text stream. That object may represent a document's entire text, or just part of it. Most applications permit the user to have several documents open at once. The user may request several checks for the same document. During a spelling check, the NXSpellChecker notes words that the user has accepted by clicking the Ignore button in the Spell Panel. To make the ignored words feature more useful, the list developed during the current check must be consolidated with ignored words noted in previous checks.

The NXIgnoreMisspelledWords protocol specifies a method by which the NXSpellChecker can ask the text it is checking, "What do you belong to?" The method **spellDocumentTag** must return a tag that the NXSpellChecker can use to distinguish the documents being checked. (See the discussion of "Matching a List of Ignored Words With the Document It Belongs To" in the description of the NXSpellChecker class.) Once the NXSpellChecker has a way to distinguish the various documents, it can append new ignored words to the appropriate list. The application can then ask the NXSpellChecker to initialize its private list of ignored words by copying them from a list stored with the document.

See also: – **setIgnoredWords:** (NXSpellChecker)

Instance Methods

spellDocumentTag

– (int)**spellDocumentTag**

Returns an arbitrary integer to identify the document (or other source) from which a text stream comes, and to distinguish between alternative documents that may be open at the same time.

The tags returned by this method are needed only during the current session. The tag doesn't need to reflect the name of the document or even whether the document has a name. The tag should survive unchanged if the user renames the file during the course of the session. The tag returned by this method is used as an argument to the methods **getIgnoredWordsForSpellDocument:** and **setIgnoredWords:forSpellDocument:**. For example, if **myChecker** is an NXSpellChecker instance and **tStream** is a text stream being checked:

```
[myChecker getIgnoredWordsForSpellDocument:[tStream spellDocumentTag]]
```

A tag of 0 is illegal.

NXNibNotification

(informal protocol)

Category Of: Object

Declared In: appkit/Application.h

Category Description

This informal protocol consists of a single method, **awakeFromNib**. It's implemented to receive a notification message that's sent after objects have been loaded from an Interface Builder archive.

Instance Methods

awakeFromNib

– **awakeFromNib**

Implemented to prepare an object for service after it has been loaded from an Interface Builder archive—a so-called "nib file." An **awakeFromNib** message is sent to each object loaded from the archive, but only if it can respond to the message, and only after all the objects in the archive have been loaded and initialized. When an object receives an **awakeFromNib** message, it's guaranteed to have all its outlet instance variables set.

When **loadNibFile:owner:** or a related method loads an Interface Builder archive into an application, each custom object from the archive is first initialized with an **init** message (**initFrame:** if the object is a kind of View). It's then more specifically initialized with the properties that it was programmed to have in Interface Builder. This part of the initialization process uses any **set***Variable***:** methods that are available (where *variable* is the name of an instance variable). Finally, after all the objects are fully initialized, they each receive an **awakeFromNib** message.

The order in which objects are loaded from the archive is not guaranteed. Therefore, it's possible for a **set***Variable***:** message to be sent to an object before its companion objects have been unarchived. For this reason, **set***Variable***:** methods should not send messages to other objects in the archive. However, messages to other objects can safely be sent from within **awakeFromNib**—when it's assured that that all the objects are unarchived and fully initialized.

Typically, **awakeFromNib** is implemented for only one object in the archive, the controlling or "owner" object for the other objects that are archived with it. For example, suppose that a nib file contained two Views that must be positioned relative to each other at run time. Trying to position them when either one of the Views is initialized (in a **set***Variable***:** method) might fail, since the other View might not be unarchived and initialized yet. However, it can be done in an **awakeFromNib** method:

```
- awakeFromNib
{
    NXRect viewFrame;

    [firstView getFrame:&viewFrame];
    [secondView moveTo:viewFrame.origin.x + someVariable
                      :viewFrame.origin.y];
    return self;
}
```

There's no default **awakeFromNib** method. The Application Kit declares a prototype for this method, but doesn't implement it.

See also: **– loadNibFile:owner:withNames:fromZone:** (Application class)

NXPrintingUserInterface

(informal protocol)

Category Of: Object

Declared In: appkit/View.h

Category Description

This informal protocol consists of one method, **shouldRunPrintPanel:**, which is implemented by initiators of printing (and faxing) requests. Its return value indicates whether the Print panel (or Fax panel) should be displayed to the user.

Instance Methods

shouldRunPrintPanel:

– (BOOL)**shouldRunPrintPanel:***aView*

Implemented to indicate whether the Print panel (or Fax panel) should be run before printing (or faxing) begins. Running the panel means placing it on-screen so the user can make choices about the print job, or possibly even cancel it. Not running the panel means that the print job proceeds without user intervention.

Printing requests are initiated by sending a View or Window object a message to perform one of these two methods:

printPSCode:	(View and Window)
smartPrintPSCode:	(Window only)

Each method takes an **id** argument, which usually identifies the initiator of the print request (the object that sent the message). A **shouldRunPrintPanel:** message is sent back to that object, if the object can respond to the message. The *aView* argument identifies the View being printed—the View that received the **printPSCode:** message. If a Window received the **printPSCode:** (or **smartPrintPSCode:**) message, *aView* is the frame view for the Window.

If **shouldRunPrintPanel:** returns YES, the Print panel is run before printing begins. If it returns NO, the Print panel is not run, and the previous settings of the panel are used. The Print panel is also run if this method is not implemented.

Requests to fax a View or a Window can be initiated (by users) from within the Print panel. However, an application can bypass the Print panel using either of the following two methods, which parallel the printing methods listed above:

faxPSCode:	(View and Window)
smartFaxPSCode:	(Window only)

Like the printing methods, these methods each take an **id** argument, and that argument is sent a **shouldRunPrintPanel:** message if it can respond. However, in this case, the value returned by **shouldRunPrintPanel:** indicates whether the Fax panel (not the Print panel) should be run.

There's no default implementation of the **shouldRunPrintPanel:** method. The Application Kit declares a prototype for this method, but doesn't define it. Therefore, there's no reason to implement this method just to return YES. If it's not implemented, the print and fax panels will be run by default.

See also: – **printPSCode:** (View and Window classes), – **smartPrintPSCode:** (Window class), – **faxPSCode:** (View and Window classes), – **smartFaxPSCode:** (Window class)

NXReadOnlyTextStream

Adopted By: Text

Declared In: appkit/readOnlyTextStream.h

Protocol Description

This is the protocol that a delegate of NXSpellServer uses to provide the text to be spell-checked. An object that implements this protocol is one of the arguments to the NXSpellChecker method that initiates a spell-checking session.

The data provided by NXReadOnlyTextStream must contain purely NeXTSTEP-encoding characters—the spell checker doesn't support multiple byte characters. Since one byte in the stream must equal one character, 16-bit entities (such as those in the KANJI system) must be converted to obvious break characters (such as space) before being handed out by this protocol. All formatting characters and other nontext characters should be completely stripped out. Only the actual one-byte NeXTSTEP-encoded characters should come through this protocol.

Note that an object that wants to use spell-checking may choose to break data up into lots of little NXReadOnlyTextStreams. One example of this is when you have multiple languages within a document in your application. You can return YES in the method **isAtEOTS** (and stop returning characters via **readCharacters:count:**) whenever you reach the end of one language and the beginning of another. You can then tell the NXSpellChecker object to switch languages and start another spell checking session on subsequent text.

Note that the **currentCharacterOffset** method should always return a value reflecting either the position of the insertion point of the text, or the start of the selection in the text, or, if neither of those exists, the start of the text. This means that the NXReadOnlyTextStream is expected to be valid only for the duration of a single invocation of **checkSpelling:of:**.

Method Types

Opening and closing the stream	– openTextStream – closeTextStream
Reading from the stream	– readCharacters:count:
Positiioning within stream	– seekToCharacterAt:relativeTo: – currentCharacterOffset – isAtEOTS

Instance Methods

closeTextStream

– **closeTextStream**

Closes the text stream, returns **self**. This method is invoked at the end of a spell-checking session.

currentCharacterOffset

– (int)**currentCharacterOffset**

Returns, at the beginning of the spell-checking session, either the insertion point of the text, or the start of the selection in the text, or, if neither of those exists, the start of the text on the stream. The value returned by this method should be updated to reflect the new position each time **readCharacters:count:** is invoked.

isAtEOTS

– (BOOL)**isAtEOTS**

Tests whether the receiver has reached the end of the text stream. Returns YES if it has, NO if not.

openTextStream

– **openTextStream**

Opens the text stream, returns **self**. This method is invoked at the beginning of a spell-checking session.

readCharacters:count:

– (int)**readCharacters:**(char *)*buffer* **count:**(int)*count*

Requests character data from the NXReadOnlyTextStream, beginning at the current position. *count* is the number of characters requested. *buffer* represents the storage into which those characters should be copied. This method returns the number of characters actually placed on the stream, which may be less than *count* if the number of characters remaining in the text is less than *count*.

seekToCharacterAt:relativeTo:

– (BOOL)**seekToCharacterAt:**(int)*offset* **relativeTo:**(int)*seekMode*

Sets the current position for reading characters from the NXReadOnlyTextStream. *offset* represents the number of characters to move, and may be negative if, for example, the seek is from the end of a stream. *seekMode* represents the position from which to begin the seek; it should be one of the constants NX_StreamStart, NX_StreamCurrent, and NX_StreamEnd. Returns YES if the current position was set successfully as specified.

NXRTFDErrorHandler

Adopted By: no NeXTSTEP classes

Declared In: appkit/NXRTFDErrors.h

Protocol Description

This protocol contains one method, **attemptOverwrite:**. An object that implements this method can intervene when a Text object has trouble saving an RTFD document.

Instance Methods

attemptOverwrite:

– (BOOL)**attemptOverwrite:**(const char *)*filename*

Notifies the receiver that the user is attempting to save an RTFD document in a location for which the user doesn't have search permission. Returning YES allows the Text object to complete the save operation; returning NO aborts it.

The Text class's **saveRTFDTo:removeBackup:errorHandler:** method takes an error handling object as its third argument: This is the object that receives the **attemptOverwrite:** message under the circumstances mentioned above. The error handler can respond to such a message by displaying a panel that asks the user whether the file should be written.

See also: – **saveRTFDTo:removeBackup:errorHandler:** (Text class)

NXSelectText

Adopted By: Text

Declared In: appkit/NXSpellChecker.h

Protocol Description

This protocol must be implemented by an object whose text is to be checked, so that it can respond to messages from an NXSpellChecker.

Instance Methods

makeSelectionVisible

– (void)**makeSelectionVisible**

Scrolls the view that contains the selection so that the selection is visible.

readCharactersFromSelection:count:

– (int)**readCharactersFromSelection:**(char *)*buffer* **count:**(int)*count*

Reads a substring from the selected portion of the text stream. The argument *count* is the desired number of characters to read. The argument *buffer* is a pointer to the start of the resulting character string. Returns the number of characters actually read (which may be smaller than the number requested if the selected portion of the text stream is not that long).

selectCharactersFrom:to:

– (void)**selectCharactersFrom:**(int)*start* **to:**(int)*end*

Selects (that is, highlights in the document's display) the specified block of characters. The arguments *start* and *end* are the number of characters from the start of the text stream (which may differ from the number of bytes if the text stream includes multibyte characters).

selectionCharacterCount

– (int)**selectionCharacterCount**

Returns the length (that is, the number of characters) of the portion of the text stream that is currently selected.

NXServicesRequests

(informal protocol)

Category Of: Object

Declared In: appkit/Application.h

Category Description

This informal protocol consists of two methods, **writeSelectionToPasteboard:types:** and **readSelectionFromPasteboard:**. The first is implemented to provide data to a remote service, and the second to receive any data the remote service might send back. Both respond to messages that are generated when the user chooses a command from the Services menu.

Instance Methods

readSelectionFromPasteboard:

– **readSelectionFromPasteboard:***pboard*

Implemented to replace the current selection with data read from the Pasteboard object *pboard*. The data would have been placed in the pasteboard by another application in response to a remote message from the Services menu. A **readSelectionFromPasteboard:** message is sent to the same object that previously received a **writeSelectionToPasteboard:types:** message.

There's no default **readSelectionFromPasteboard:** method. The Application Kit declares a prototype for this method, but doesn't implement it.

See also: – **writeSelectionToPasteboard:types:**

writeSelectionToPasteboard:types:

– (BOOL)**writeSelectionToPasteboard:***pboard* **types:**(NXAtom *)*types*

Implemented to write the current selection to the Pasteboard object *pboard*. The selection should be written as one or more of the data types listed in *types*. After writing the data, this method should return YES. If for any reason it can't write the data, it should return NO.

A **writeSelectionToPasteboard:types:** message is sent to the first responder when the user chooses a command from the Services menu, but only if the receiver didn't return **nil** to a previous **validRequestorForSendType:andReturnType:** message.

After this method writes the data to the pasteboard, a remote message is sent to the application that provides the service the user requested. If the service provider supplies return data to replace the selection, the first responder will then receive a **readSelectionFromPasteboard:** message.

There's no default **writeSelectionToPasteboard:types:** method. The Application Kit declares a prototype for this method, but doesn't implement it.

See also: – **validRequestorForSendType:andReturnType:** (Responder class), – **readSelectionFromPasteboard:**

NXWorkspaceRequestProtocol

Adopted By: No NeXTSTEP classes

Declared In: appkit/workspaceRequest.h

Protocol Description

The **NXWorkspaceRequestProtocol** protocol is implemented by an object in the Workspace Manager that responds to application requests to do such things as open files, launch applications, and return file icons. This object is made available through Application's **workspace** method. As an example, the following code uses the Workspace Manager object to request that a file be opened in the Edit application:

```
[[Application workspace] openFile:"/tmp/README"
    withApplication:"Edit"];
```

Before NeXTSTEP Release 3, some of the functionality of the NXWorkspaceRequestProtocol protocol was found in the Speaker and Listener classes. New applications should be changed to send messages to the Workspace Manager object rather than to Workspace Manager's Listener.

Many of the methods in the NXWorkspaceRequestProtocol protocol depend on the existence of an Application object and its Speaker and/or Listener objects. This presents no problem for NeXTSTEP applications, but other applications may have to create an Application object and send it a **run** message in order to use these methods.

Method Types

Opening files
- openFile:
- openFile:withApplication:
- openFile:fromImage:at:inView:
- openFile:withApplication:andDeactivate:
- openTempFile:
- findString:inFile:

Manipulating applications
- launchApplication:
- launchApplication:showTile:autolaunch:
- hideOtherApplications

Manipulating files
- performFileOperation:source:destination:files:options:
- selectFile:inFileViewerRootedAt:

Requesting information about files
- getIconForFile:
- getInfoForFile:application:type:
- getFullPathForApplication:
- getInfoForFileSystemAt:isRemovable:isWritable: isUnmountable:description:type:

Requesting additional time before logout
- extendPowerOffBy:

Tracking changes to the file system
- fileSystemChanged
- didFileSystemChange

Updating registered services and file types
- findApplications

Tracking changes to the defaults database
- defaultsChanged
- didDefaultsChange

Tracking status changes for applications and devices
- beginListeningForApplicationStatusChanges
- endListeningForApplicationStatusChanges
- beginListeningForDeviceStatusChanges
- endListeningForDeviceStatusChanges

Animating an image
- slideImage:from:to:

Unmounting a device
- unmountAndEjectDeviceAt:

Instance Methods

beginListeningForApplicationStatusChanges

– (void)**beginListeningForApplicationStatusChanges**

Notifies Workspace Manager that the application wants to be notified of changes in the status of all applications. After sending this message, the Application object's delegate will receive the following messages when an application is launched or after one terminates:

app:applicationWillLaunch:
app:applicationDidLaunch:
app:applicationDidTerminate:

See also: – **endListeningForApplicationStatusChanges**

beginListeningForDeviceStatusChanges

– (void)**beginListeningForDeviceStatusChanges**

Notifies Workspace Manager that the application wants to be notified when various media (usually optical or floppy disks) are mounted or unmounted. After sending this message, the Application object's delegate will receive the following messages after a device is mounted or unmounted:

app:mounted:
app:unmounted:

These methods complement the **unmounting:ok:** method, which is sent just before a device is unmounted so that applications can end all their accesses to that device.

See also: – **endListeningForDeviceStatusChanges**

defaultsChanged

– (void)**defaultsChanged**

Informs Workspace Manager that the defaults database has changed. Workspace Manager then reads all the defaults it is interested in and reconfigures itself appropriately. For example, this method is used by the Preferences application to notify Workspace Manager whether the user prefers to see hidden files.

See also: – **didDefaultsChange**

didDefaultsChange

– (BOOL)**didDefaultsChange**

Returns whether a change to the defaults database has been registered with a **defaultsChanged** message, and clears the internal flag (for the sending application) that indicates such a change.

didFileSystemChange

– (BOOL)**didFileSystemChange**

Returns whether a change to the file system has been registered with a **fileSystemChanged** message, and clears the internal flag (for the sending application) that indicates such a change.

endListeningForApplicationStatusChanges

– (void)**endListeningForApplicationStatusChanges**

Notifies Workspace Manager that the application is no longer interested in notifications of application launches and terminations.

See also: – **beginListeningForApplicationStatusChanges**

endListeningForDeviceStatusChanges

– (void)**endListeningForDeviceStatusChanges**

Notifies Workspace Manager that the application is no longer interested in notifications of the mounting and unmounting of devices (such as optical and floppy disks).

See also: – **beginListeningForDeviceStatusChanges**

extendPowerOffBy:

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

Requests more time before the power goes off or the user logs out. An application can send this message 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 granted number of additional milliseconds is returned.

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

fileSystemChanged

– (void)**fileSystemChanged**

Informs Workspace Manager that the file system has changed. Workspace Manager then gets the status of all the files and directories it is interested in and updates itself appropriately. This method is used by many objects that write or delete files. Even if this method isn't invoked, Workspace Manager will note changes to the file system relatively quickly if it is the active application.

See also: – **didFileSystemChange**

findApplications

– (void)**findApplications**

Instructs Workspace Manager to examine all applications in the normal places and update its records of registered services and file types. This can be useful in an project building application, for example.

findString:inFile:

– (BOOL)**findString:**(const char *)*aString* **inFile:**(const char *)*filename*

Instructs Workspace Manager to open the file *filename* (specified with a complete path), with the string *aString* selected. The file is opened using the default application for its type. The application that opens the file must implement the **msgSetPosition:posType:andSelect:ok:** message in its Listener's delegate to find the selection.

getFullPathForApplication:

– (const char *)**getFullPathForApplication:**(const char *)*appName*

Returns the full path for the application *appName*, or NULL if *appName* isn't in one of Workspace Manager's application directories. The returned string is valid until the next message to a Speaker object; the application must copy it if it must be retained.

getIconForFile:

– (NXImage *)**getIconForFile:**(const char *)*fullPath*

Returns a newly allocated NXImage with the icon for the single file specified by *fullPath*, or **nil** if there is an error.

See also: – **getInfoForFile:application:type:**

getInfoForFile:application:type:

– (BOOL)**getInfoForFile:**(const char *)*fullPath*
application:(char **)*appName*
type:(NXAtom *)*type*

Retrieves information about the file specified by *fullPath*. After invoking this method, the string pointed to by *appName* is set to the application Workspace Manager would use to open *fullPath*. This string should be freed by the caller. The NXAtom pointed to by *type* will contain one of the following values or a file name extension such as "rtf" indicating the file's type:

Value	***fullPath* is a:**
NXPlainFileType	plain (untyped) file
NXDirectoryFileType	directory
NXApplicationFileType	NeXTSTEP application
NXFilesystemFileType	file system mount point
NXShellCommandFileType	executable shell command

Returns YES upon success, NO otherwise.

See also: – **getIconForFile:**

getInfoForFileSystemAt:isRemovable:isWritable:isUnmountable: description:type:

– (BOOL)**getInfoForFileSystemAt:**(const char *)*fullPath*
isRemovable:(BOOL *)*removableFlag*
isWritable:(BOOL *)*writableFlag*
isUnmountable:(BOOL *)*unmountableFlag*
description:(char **)*description*
type:(char **)*fileSystemType*

Describes the file system at *fullPath*. Returns YES if *fullPath* is a file system mount point, or NO if it isn't. Upon success, *description* will describe the file system; this value can be used in strings, but it shouldn't be depended upon by program logic. Example values for *description* are "hard", "nfs", and "foreign". *fileSystemType* will indicate the file system type; values could be "NeXT", "DOS", or other values.

hideOtherApplications

– (void)**hideOtherApplications**

Hides all applications other than the sender. (Since the user can cause the same effect by Command-double-clicking on an application's tile, a programmatic invocation of this method is usually unnecessary.)

launchApplication:

– (BOOL)**launchApplication:**(const char *)*appName*

Instructs Workspace Manager to launch the application *appName*. *appName* need not be specified with a full path and, in the case of an application wrapper, can be specified with or without the ".app" extension. Returns YES if the application is successfully launched or already running, and NO if it can't be launched.

See also: – launchApplication:showTile:autolaunch:

launchApplication:showTile:autolaunch:

– (BOOL)**launchApplication:**(const char *)*appName*
showTile:(BOOL)*showTile*
autolaunch:(BOOL)*autolaunch*

Instructs Workspace Manager to launch the application *appName*. If *showTile* is NO, Workspace Manager won't display a tile for the application. (The tile will exist, but it won't be placed on the screen.) If *autolaunch* is YES, the NXAutoLaunch command line default will be set as though the application were autolaunched from the dock. This method is provided to enable daemon-like apps that lack a normal user interface and for use by alternative dock programs. Its use is not generally encouraged.

Returns YES if the application is successfully launched or already running, and NO if it can't be launched.

See also: – **launchApplication:**

openFile:

– (BOOL)**openFile:**(const char *)*fullPath*

Instructs Workspace Manager to open the file specified by *fullPath* using the default application for its type. The sending application is deactivated before the request is sent. Returns YES if the file is successfully opened, and NO otherwise.

See also: – **openFile:withApplication:andDeactivate:**, – **openTempFile:**

openFile:fromImage:at:inView:

– (BOOL)**openFile:**(const char *)*fullPath*
fromImage:(NXImage *)*anImage*
at:(const NXPoint *)*point*
inView:(View *)*aView*

Instructs Workspace Manager to open the file specified by *fullPath* using the default application for its type. Before opening the file, Workspace Manager will provide animation to give the user feedback that the file is to be opened. To provide this animation, *anImage* should contain an icon for the file, and its image should be displayed at *point*, specified in *aView*'s coordinates.

The sending application is deactivated before the request is sent. Returns YES if the file is successfully opened, and NO otherwise.

See also: – **openFile:withApplication:andDeactivate:**

openFile:withApplication:

– (BOOL)**openFile:**(const char *)*fullPath* **withApplication:**(const char *)*appName*

Instructs Workspace Manager to open the file specified by *fullPath* using the *appName* application. *appName* need not be specified with a full path and, in the case of an application wrapper, can be specified with or without the ".app" extension. The sending application is deactivated before the request is sent. Returns YES if the file is successfully opened, and NO otherwise.

See also: – **openFile:withApplication:andDeactivate:**

openFile:withApplication:andDeactivate:

– (BOOL)**openFile:**(const char *)*fullPath*
withApplication:(const char *)*appName*
andDeactivate:(BOOL)*flag*

Instructs Workspace Manager to open the file specified by *fullPath* using the *appName* application. *appName* need not be specified with a full path and, in the case of an application wrapper, can be specified with or without the ".app" extension. If *appName* is NULL, the default application for the file's type is used. If *flag* is YES, the sending application is deactivated before the request is sent, allowing the opening application to become the active application. Returns YES if the file is successfully opened, and NO otherwise.

See also: – **app:openFile:type:** (Application delegate)

openTempFile:

– (BOOL)**openTempFile:**(const char *)*fullPath*

Instructs Workspace Manager to open the temporary file specified by *fullPath* using the default application for its type. The sending application is deactivated before the request is sent. Using this method instead of one of the **openFile:...** methods lets the receiving application know that it should delete the file when it no longer needs it. Returns YES if the file is successfully opened, and NO otherwise.

See also: – **openFile:withApplication:andDeactivate:**

performFileOperation:source:destination:files:options:

– (int)**performFileOperation:**(const char *)*operation*
source:(const char *)*source*
destination:(const char *)*destination*
files:(const char *)*files*
options:(const char *)*options*

Requests that the Workspace Manager perform some file operation, such as copying or moving files. The files to be manipulated are specified by *files*. If more than one file is specified, the files must be tab-delimited. This list is specified relative to the source directory, *source*. The list can contain both files and directories; all of them must be directly under *source* (not under one of its subdirectories). Some operations require a destination directory, *destination*; otherwise it should be the empty string (""). The permissible values for *operation* are as follows:

Operation	Meaning
WSM_MOVE_OPERATION	Move file to destination
WSM_COPY_OPERATION	Copy file to destination
WSM_LINK_OPERATION	Create link to file in destination
WSM_COMPRESS_OPERATION	Compress file
WSM_DECOMPRESS_OPERATION	Decompress file
WSM_ENCRYPT_OPERATION	Encrypt file
WSM_DECRYPT_OPERATION	Decrypt file
WSM_DESTROY_OPERATION	Destroy file
WSM_RECYCLE_OPERATION	Move file to recycler
WSM_DUPLICATE_OPERATION	Duplicate file in source directory

Note: WSM_ENCRYPT_OPERATION and WSM_DECRYPT_OPERATION might not be available on all systems.

Returns a negative integer if the operation fails, 0 if the operation is performed synchronously and succeeds, and a positive integer if the operation is performed asynchronously. The positive integer is a tag that will be sent to the application using the **app:fileOperationCompleted:** Application delegate method when the operation completes.

selectFile:inFileViewerRootedAt:

– (BOOL)**selectFile:**(const char *)*fullPath*
inFileViewerRootedAt:(const char *)*rootFullpath*

Instructs Workspace Manager to select the file specified by *fullPath*. If a path is specified by *rootFullpath*, a new file viewer is opened. If *rootFullpath* is an empty string (“”), the file is selected in the main viewer. Returns YES if the file is successfully selected, and NO otherwise.

slideImage:from:to:

– (void)**slideImage:**(NXImage *)*image*
from:(const NXPoint *)*fromPoint*
to:(const NXPoint *)*toPoint*

Instructs Workspace Manager to animate a sliding image of *image* from *fromPoint* to *toPoint*, specified in screen coordinates.

unmountAndEjectDeviceAt:

– (BOOL)**unmountAndEjectDeviceAt:**(const char *)*path*

Unmounts and ejects the device at *path*. Returns YES if the device is successfully unmounted or *path* is badly formed; returns NO otherwise.

Functions

NXAlphaComponent() → See NXRedComponent()

NXAttachPopUpList(), NXCreatePopUpListButton()

SUMMARY Set up a pop-up list

DECLARED IN appkit/PopUpList.h

SYNOPSIS void **NXAttachPopUpList**(id *button*, PopUpList **popUpList*)
id **NXCreatePopUpListButton**(PopUpList **popUpList*)

DESCRIPTION These functions make it easy to use the PopUpList class. **NXCreatePopUpListButton()** returns a new Button object that will activate the pop-up list specified by *popUpList*. The new Button must then be added to the View hierarchy with View's **addSubview:** method.

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. *button* must be either a Control that uses ButtonCell (or a subclass) as its Cell class, or an actual ButtonCell.

RETURN **NXCreatePopUpListButton()** returns a new Button object.

NXBeep()

SUMMARY Play the system beep

DECLARED IN appkit/publicWraps.h

SYNOPSIS 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

DECLARED IN appkit/timer.h

SYNOPSIS 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()

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

DECLARED IN appkit/color.h

SYNOPSIS

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

DECLARED IN appkit/chunk.h

SYNOPSIS
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 are 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()

NXColorListName(), NXColorName(), NXFindColorNamed()

SUMMARY Associate colors with their names and their color lists

DECLARED IN appkit/color.h

SYNOPSIS
const char *__NXColorListName__ (NXColor *color*)
const char *__NXColorName__ (NXColor *color*)
BOOL __NXFindColorNamed__ (const char **colorList*, const char **colorName*, NXColor **color*)

DESCRIPTION Use these functions to access named colors and named color lists. They're used in conjunction with objects of the NXColorList class that generate colors with persistent names. The documentation for NXColorList includes a complete description of persistent color names and named NXColorLists.

NXColorListName() looks for and returns the name of a color list from which a particular color was taken. *color* represents the NXColor whose source list name you're seeking. The return value is a character string representing the name of the list from which the color was taken; an empty string is returned if *color* isn't taken from a named list. This function can be used to get the argument for NXColorList's **findColorListNamed:** method.

NXColorName() returns the persistent name of a color. *color* is the NXColor whose persistent name you wish to find.

NXFindColorNamed() returns by reference the NXColor associated with a particular name in a particular list. *colorList* represents the name of the list in which you wish to search; if *colorList* doesn't represent a NXColorList that generates colors with persistent names, this method returns NO. *colorName* represents the name of the color you wish to find. *color* returns the actual NXColor associated with *colorName* in *colorList*.

RETURN **NXColorListName()** returns a character string representing the name of the list from which the color was taken; an empty string is returned if the color isn't taken from a named list.

NXColorName() returns a character string representing the name of *color*: an empty string is returned if *color* wasn't taken from a list that generates colors with persistent names.

NXFindColorNamed() returns YES if it finds *colorName* in *colorList*, NO if not.

SEE ALSO NXColorList class

NXColorName() → See NXColorListName()

NXColorSpaceFromDepth(), NXBPSFromDepth(), NXNumberOfColorComponents(), NXGetBestDepth()

SUMMARY Get information about color space and window depth

DECLARED IN appkit/graphics.h

SYNOPSIS
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.

NXCompleteFilename()

SUMMARY Match an incomplete file name

DECLARED IN appkit/SavePanel.h

SYNOPSIS 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*).

RETURN 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 NXConvertColorToRGBA()

NXConvertColorToCMYKA() → See NXConvertColorToRGBA()

NXConvertColorToGray() → See NXConvertColorToRGBA()

NXConvertColorToGrayAlpha() → See NXConvertColorToRGBA()

NXConvertColorToHSB() → See NXConvertColorToRGBA()

NXConvertColorToHSBA() → See NXConvertColorToRGBA()

NXConvertColorToRGB() → See NXConvertColorToRGBA()

NXConvertColorToRGBA(), NXConvertColorToCMYKA(), NXConvertColorToHSBA(), NXConvertColorToGrayAlpha(), NXConvertColorToRGB(), NXConvertColorToCMYK(), NXConvertColorToHSB(), NXConvertColorToGray()

SUMMARY Convert a color value to its standard components

DECLARED IN appkit/color.h

SYNOPSIS

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

DECLARED IN appkit/color.h

SYNOPSIS

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

DECLARED IN appkit/publicWraps.h

SYNOPSIS 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 access to 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(), NXCopyBitmapFromGstate()

SUMMARY Copy an image

DECLARED IN appkit/graphics.h

SYNOPSIS
void **NXCopyBits**(int *gstate*, const NXRect **aRect*, const NXPoint **aPoint*)
void **NXCopyBitmapFromGstate**(int *gstate*, const NXRect **srcRect*, const NXRect **destRect*)

DESCRIPTION **NXCopyBits**() copies 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 *gstate*. If *gstate* is NXNullObject, the current graphics state is assumed. The *aPoint* destination is defined in the current graphics state.

NXCopyBitmapFromGstate() copies the pixels in the rectangle *srcRect* to the rectangle *destRect*. The source rectangle is defined in the graphics state designated by *gstate*. The destination is defined in the current graphics state.

NXCopyCurrentGState() → See NXSetGState()

NXCopyInputData(), NXCopyOutputData()

SUMMARY Save data received in a remote message

DECLARED IN appkit/Listener.h

SYNOPSIS
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 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()

NXCountWindows(), NXWindowList()

SUMMARY Get information about an application's windows

DECLARED IN appkit/publicWraps.h

SYNOPSIS 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()**.

NXCreateFileContentsPboardType(), NXCreateFilenamePboardType(), NXGetFileType(), NXGetFileTypes()

SUMMARY Return file-related pasteboard types

DECLARED IN appkit/Pasteboard.h

SYNOPSIS
NXAtom **NXCreateFileContentsPboardType**(const char ***fileType*)
NXAtom **NXCreateFilenamePboardType**(const char ***filename*)
const char ***NXGetFileType**(const char ***pboardType*)
const char ****NXGetFileTypes**(const char *const **pboardTypes*)

DESCRIPTION **NXCreateFileContentsPboardType()** returns an NXAtom to a pasteboard type representing a file's contents based on the supplied string *fileType*. *fileType* should generally be the extension part of a file name. The conversion from a named file type to a pasteboard type is simple; no mapping to standard pasteboard types is attempted.

NXCreateFilenamePboardType() returns an NXAtom to a pasteboard type representing a file name based on the supplied string *filename*.

NXGetFileType() is the inverse of both **NXCreateFileContentsPboardType()** and **NXCreateFilenamePboardType()**. When passed a pasteboard type as returned by those functions, it returns the extension or file name from which the type was derived. It returns NULL if *pboardType* isn't a pasteboard type created by those functions.

NXGetFileTypes() accepts a null-terminated array of pointers to pasteboard types and returns a null-terminated array of the unique extensions and file names from the file-content and file-name types found in the input array. It returns NULL if the input array contains no file-content or file-name types. The returned array is allocated and must be freed by the caller. The pointers in the return array point into strings passed in the input array.

NXCreatePopUpListButton() → See NXAttachPopUpList()

NXCyanComponent() → See NXRedComponent()

NXDefaultStringOrderTable() → See NXOrderStrings()

NXDefaultTopLevelErrorHandler(), NXSetTopLevelErrorHandler(), NXTopLevelErrorHandler()

SUMMARY Define an error handler

DECLARED IN appkit/errors.h

SYNOPSIS

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 (as defined in the header file **objc/error.h**). The **appkit/errors.h** header file 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**() (Common Functions), **NXDefaultExceptionRaiser**() (Common Functions), **NXRegisterErrorReporter**()

NXDivideRect() → See NXSetRect()

NXDrawALine() → See NXScanALine()

NXDrawBitmap(), NXReadBitmap(), NXSizeBitmap()

SUMMARY Render and read bitmap images

DECLARED IN appkit/graphics.h

SYNOPSIS void **NXDrawBitmap**(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_)

Warning: These functions are marginally obsolete. Most applications are better served using the NXBitmapImageRep class to read and display bitmap images.

DESCRIPTION The first of these functions, **NXDrawBitmap()**, 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 **NXDrawBitmap()** can use to recreate the image. The third function, **NXSizeBitmap()**, supplies the information required by **NXReadBitmap()**.

NXDrawBitmap() 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 **NXDrawBitmap()** 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 2-2 illustrates these two ways of configuring data.

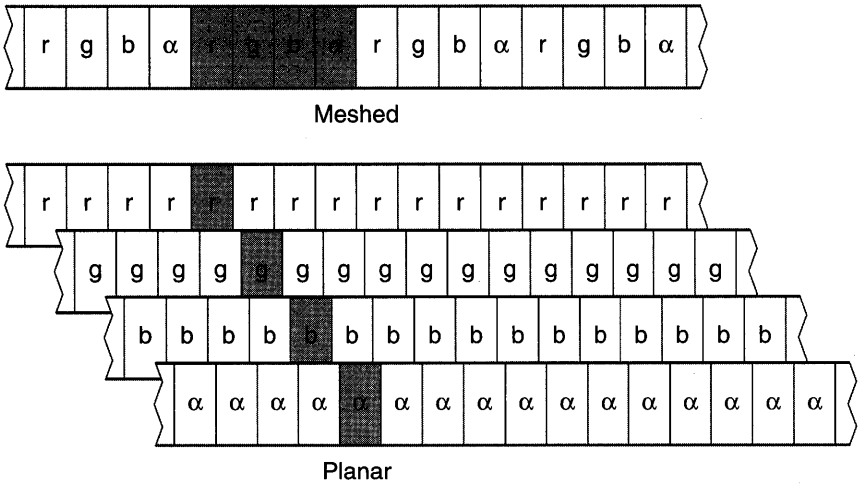


Figure 2-2. Planar and Meshed Configurations

As shown in the illustration, color samples (rgb) precede the coverage sample (α) in both configurations.

In NeXTSTEP, gray-scale windows store pixel data in planar configuration; color windows store it in meshed configuration. **NXDrawBitmap()** 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). NeXTSTEP uses 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.

NXDrawButton(), NXDrawGrayBezel(), NXDrawGroove(), NXDrawWhiteBezel(), NXDrawTiledRects(), NXFrameRect(), NXFrameRectWithWidth()

SUMMARY Draw a bordered rectangle

DECLARED IN appkit/graphics.h

SYNOPSIS

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 in the NeXTSTEP user interface, **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 2-3.

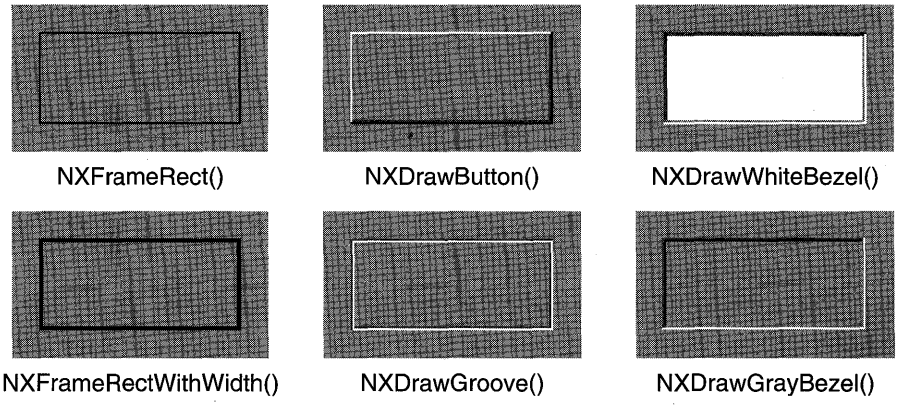


Figure 2-3. Rectangle Borders

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()

NXEmptyRect() → See NXMouseInRect()

NXEndTimer() → See NXBeginTimer()

NXEqualColor()

SUMMARY Test whether two colors are the same

DECLARED IN appkit/color.h

SYNOPSIS 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

DECLARED IN appkit/Text.h

SYNOPSIS 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.

NXFindColorNamed() → See NXColorListName()

NXFindPaperSize()

SUMMARY Find dimensions of specified paper type

DECLARED IN appkit/PageLayout.h

SYNOPSIS 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). *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.

NXFrameLinkRect(), NXLinkFrameThickness()

SUMMARY Draw a distinctive outline around linked data

DECLARED IN appkit/NXDataLinkManager.h

SYNOPSIS
void **NXFrameLinkRect**(const NXRect *_aRect_, BOOL _isDestination_)
float **NXLinkFrameThickness**(void)

DESCRIPTION **NXFrameLinkRect()** draws a distinctive link outline just outside the rectangle specified by _aRect_. To draw an outline around a destination link, _isDestination_ should be YES, otherwise it should be NO. **NXLinkFrameThickness()** returns the thickness of the link outline so that the outline can be properly erased by the application, or for other purposes.

NXFrameRect() → See NXDrawButton()

NXFrameRectWithWidth() → See NXDrawButton()

NXFreeAlertPanel() → See NXRunAlertPanel()

NXGetAlertPanel() → See NXRunAlertPanel()

NXGetBestDepth() → See NXColorSpaceFromDepth()

NXGetFileType() → See NXCreateFileContentsPboardType()

NXGetFileTypes() → See NXCreateFileContentsPboardType()

NXGetNamedObject(), NXGetObjectName(), NXNameObject(), NXUnnameObject()

SUMMARY Refer to objects by name

DECLARED IN appkit/Application.h

SYNOPSIS
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 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 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 **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

DECLARED IN appkit/Application.h

SYNOPSIS 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 through Application's **context** method. 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. See "Types and Constants" for a list of event type masks.

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()** (Display PostScript), **DPSPeekEvent()** (Display PostScript), **DPSDiscardEvent()** (Display PostScript)

NXGetWindowServerMemory()

SUMMARY Return the amount of memory being used by a context

DECLARED IN appkit/Application.h

SYNOPSIS int **NXGetWindowServerMemory**(DPSContext *context*, int **virtualMemory*, int **windowBackingMemory*, 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 *virtualMemory*; the amount of window backing store used by windows owned by the current context is returned in the **int** pointed to by *windowBackingMemory*. The sum of these two numbers is the amount of the Window Server's memory that this context is responsible for.

To calculate these numbers, **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 specified NXStream. 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, this function returns –1.

NXGrayComponent() → See NXRedComponent()

NXGreenComponent() → See NXRedComponent()

NXHighlightRect() → See NXRectClip()

NXHomeDirectory(), NXUserName()

SUMMARY Get user's home directory and name

DECLARED IN appkit/Application.h

SYNOPSIS
const char ***NXHomeDirectory**(void)
const char ***NXUserName**(void)

DESCRIPTION These functions return the user's home directory and name.

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()

NXInsetRect() → See NXSetRect()

NXIntegralRect() → See NXSetRect()

NXIntersectionRect() → See NXUnionRect()

NXIntersectsRect() → See NXMouseInRect(

NXIsServicesMenuItemEnabled() → See NXSetServicesMenuItemEnabled()

NXJournalMouse()

SUMMARY Allow journaling during direct mouse tracking

DECLARED IN appkit/NXJournaler.h

SYNOPSIS 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, 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**()

NXLinkFrameThickness() → See NXFrameLinkRect()

NXLogError()

SUMMARY Write a formatted error string

DECLARED IN appkit/nextstd.h

SYNOPSIS void **NXLogError**(const char **format*, ...)

DESCRIPTION **NXLogError()** writes a formatted string to the console or **stderr**, depending on whether the application was launched from the Workspace Manager or a 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 **NXRegisterErrorReporter()**, **NX_RAISE()** (Common Functions), **NXDefaultExceptionRaiser()** (Common Functions)

NXMagentaComponent() → See NXRedComponent()

NXMouseInRect(), NXPointInRect(), NXIntersectsRect(), NXContainsRect(), NXEqualRect(), NXEmptyRect()

SUMMARY Test graphic relationships

DECLARED IN appkit/graphics.h

SYNOPSIS
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**(const 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.

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 NeXTSTEP 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.

SEE ALSO **NXUnionRect(), NXSetRect()**

NXNameObject() → See NXGetNamedObject()

NXNumberOfColorComponents() → See NXColorSpaceFromDepth()

NXOffsetRect() → See NXSetRect()

NXOrderStrings(), NXDefaultStringOrderTable()

SUMMARY Provide table-driven string ordering service

DECLARED IN appkit/Text.h

SYNOPSIS
int **NXOrderStrings**(const unsigned char ***string1*, const unsigned char ***string2*,
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 you can obtain 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 such as **NXIsAlpha()**, which are described in Chapter 3, "Common Classes and Functions"). 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.

NXPerformService()

SUMMARY Programmatically invokes a Services menu service

DECLARED IN appkit/Listener.h

SYNOPSIS BOOL **NXPerformService**(const char **itemName*, Pasteboard **pboard*)

DESCRIPTION **NXPerformService()** allows an application to programmatically invoke a service found in its services menu. *itemName* is a Services menu item, in any language. If the requested service is from a submenu of the Services menu, *itemName* must contain a slash (for example, "Mail/Selection"). The Pasteboard *pboard* must contain the data required by the service, and when the function returns, *pboard* will contain the data supplied by the service provider.

RETURN Returns YES if the service is successfully performed, NO otherwise.

NXPing()

SUMMARY Synchronize the application with the Window Server

DECLARED IN appkit/graphics.h

SYNOPSIS 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.

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 event 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()** (Display PostScript)

NXPointInRect() → See NXMouseInRect()

NXPortFromName(), NXPortNameLookup()

SUMMARY Get send rights to an application port

DECLARED IN appkit/Listener.h

SYNOPSIS
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". To get the port for Workspace, you should use the name NX_WORKSPACEREQUEST. Note, however, that this port isn't available until the application is fully initialized; requests for this port before Application's **appDidInit:** method is invoked will return PORT_NULL.

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 might "check in" with another server—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()

NXReadBitmap() → See NXDrawBitmap()

NXReadColor(), NXWriteColor()

SUMMARY Read and write a color from a typed stream

DECLARED IN appkit/color.h

SYNOPSIS
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.

EXCEPTIONS **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()**

NXReadColorFromPasteboard(), NXWriteColorToPasteboard()

SUMMARY Read and write NXColor data on the pasteboard

DECLARED IN appkit/color.h

SYNOPSIS NXColor **NXReadColorFromPasteboard**(id *pasteboard*)
void **NXWriteColorToPasteboard**(id *pasteboard*, NXColor *color*)

DESCRIPTION Use these functions to read and write NXColor data on a pasteboard

NXReadColorFromPasteboard() looks at *pasteboard* to see if it contains data of NXColorPboardType. If it finds color, it then checks to see if the application can import alpha, and, if not, removes any alpha component before returning the NXColor.

NXWriteColorToPasteboard() writes the NXColor *color* to the Pasteboard object *pasteboard.*

RETURN **NXReadColorFromPasteboard**() returns the color found on the pasteboard; if no color is found, it returns NX_COLORBLACK.

SEE ALSO Pasteboard class, **– doesImportAlpha** (Application class)

NXReadPixel()

SUMMARY Read and write NXColor data on the pasteboard

DECLARED IN appkit/graphics.h

SYNOPSIS NXColor **NXReadPixel**(const NXPoint **location*)

DESCRIPTION **NXReadPixel()** returns the color of the pixel at the given location. The *location* argument is taken in the current coordinate system—in other words, you must lock focus on the View that contains the pixel that you wish to query, and then pass the coordinate for the pixel in the View's coordinate system.

NXReadPoint(), NXWritePoint(), NXReadRect(), NXWriteRect(), NXReadSize(), NXWriteSize()

SUMMARY Read or write NeXTSTEP-defined data types to a typed stream

DECLARED IN appkit/graphics.h

SYNOPSIS
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 an open typed stream. They can be used within **read:** or **write:** methods for archiving purposes.

NXReadPoint(), **NXReadSize**(), and **NXReadRect**() each take a typed stream as a first argument and place the data read from the stream into the location specified by the second argument.

NXWritePoint(), **NXWriteSize**(), and **NXWriteRect**() write the data pointed to by their second arguments to the typed streams.

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**() (Common Functions), **NXReadType**() (Common Functions), **NXReadArray**() (Common Functions), **NXReadObject**() (Common Functions)

NXReadRect() → See NXReadPoint()

NXReadSize() → See NXReadPoint()

NXReadWordTable(), NXWriteWordTable()

SUMMARY Read or write Text object's word tables

DECLARED IN appkit/Text.h

SYNOPSIS

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, **NXWriteWordTable()** 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, **NXWriteWordTable()** 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 be either 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.

NXRectClip(), NXRectClipList(), NXRectFill(), NXRectFillList(), NXRectFillListWithGrays(), NXEraseRect(), NXHighlightRect()

SUMMARY Optimize drawing

DECLARED IN appkit/graphics.h

SYNOPSIS
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

DECLARED IN appkit/color.h

SYNOPSIS
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 function 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**()

NXRegisterErrorReporter(), NXRemoveErrorReporter(), NXReportError()

SUMMARY Specify an error reporter

DECLARED IN appkit/errors.h

SYNOPSIS
void **NXRegisterErrorReporter**(int *min*, int *max*, NXErrorReporter **proc*)
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 is 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 **objc/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 define 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()** (Common Functions), **NXDefaultTopLevelErrorHandler()**

NXRemoteMethodFromSel(), NXResponsibleDelegate()

SUMMARY Match an Objective C method and a receiver to a remote message

DECLARED IN appkit/Listener.h

SYNOPSIS
NXRemoteMethod ***NXRemoteMethodFromSel**(SEL *aSelector*,
NXRemoteMethod **methods*)
id **NXResponsibleDelegate**(id *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, the delegate of *aListener*'s delegate is returned. If not and *aListener*'s delegate can respond to *aSelector* messages, *aListener*'s delegate is returned. If neither delegate responds to *aSelector* messages (or *aListener* doesn't have a delegate), **nil** is returned.

NXRemoveErrorReporter() → See NXRegisterErrorReporter()

NXReportError() → See NXRegisterErrorReporter()

NXResetUserAbort() → See NXUserAborted()

NXResponsibleDelegate() → See NXRemoteMethodFromSel()

NXRunAlertPanel(), NXRunLocalizedAlertPanel(), NXGetAlertPanel(), NXFreeAlertPanel()

SUMMARY Create or free an attention panel

DECLARED IN appkit/Panel.h

SYNOPSIS

int **NXRunAlertPanel**(const char ***title*, const char ***msg*, const char ***defaultButton*, const char ***alternateButton*, const char ***otherButton*, ...)

int **NXRunLocalizedAlertPanel**(const char ***table*, 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**(), **NXRunLocalizedAlertPanel**() and **NXGetAlertPanel**() all 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**() and **NXRunLocalizedAlertPanel**() create the panel and run it in a modal event loop; **NXGetAlertPanel**() returns a Panel object 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 a Panel that can be used to set up a modal session. A modal session is useful for allowing the user to interrupt the program. During a modal session, you can perform activities while the panel is displayed and check at various points in your program whether the user has clicked one of the panel's buttons.

To set up a modal session, send the Application object a **beginModalSession:for:** message with the Panel 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 passing it to **NXFreeAlertPanel**().

RETURN **NXRunAlertPanel**() returns a constant that indicates which button in the attention panel the user clicked.

NXRunLocalizedAlertPanel() → See NXRunAlertPanel()

NXSaturationComponent() → See NXRedComponent()

NXScanALine(), NXDrawALine()

SUMMARY Calculate or draw line of text (in Text object)

DECLARED IN appkit/Text.h

SYNOPSIS
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 the Text object itself. The second argument is an NXLayInfo structure, as described in the "Types and Constants" section.

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.

NXSetColor()

SUMMARY Set the current color

DECLARED IN appkit/color.h

SYNOPSIS 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()**

NXSetGState(), NXCopyCurrentGState()

SUMMARY Set or copy current graphics state object

DECLARED IN appkit/publicWraps.h

SYNOPSIS
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

DECLARED IN appkit/graphics.h

SYNOPSIS
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 2-4.

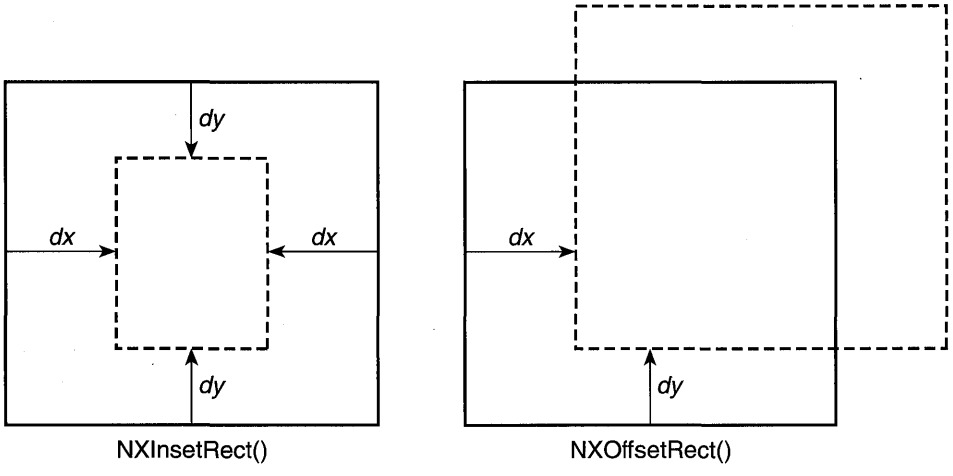


Figure 2-4. Inset and Offset Rectangles

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 constants for *edge* can be:

NX_XMIN	The slice is made parallel to the y-axis, along the side with the smallest x-coordinate values.
NX_YMIN	The slice is made parallel to the x-axis, along the side with the smallest y-coordinate values.
NX_XMAX	The slice is made parallel to the y-axis, along the side with the greatest x-coordinate values.
NX_YMAX	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

DECLARED IN appkit/Listener.h

SYNOPSIS 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 file.

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. **NXIsServicesMenuItemEnabled()** returns YES if *item* is currently enabled, and NO if it's not.

NXSetTopLevelErrorHandler() → See NXDefaultTopLevelErrorHandler()

NXSizeBitmap() → See NXDrawBitmap()

NXTextFontInfo()

SUMMARY Calculate font ascender, descender, and line height

DECLARED IN appkit/Text.h

SYNOPSIS void **NXTextFontInfo**(id *font*, NXCoord **ascender*, NXCoord **descender*, NXCoord **lineHeight*)

DESCRIPTION **NXTextFontInfo()** calculates, and returns by reference, the ascender, descender, and line height values for the Font given by *font.*

NXTopLevelErrorHandler() → See NXDefaultTopLevelErrorHandler()

NXUnionRect(), NXIntersectionRect()

SUMMARY Compute a third rectangle from two rectangles

DECLARED IN appkit/graphics.h

SYNOPSIS
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—the rectangle that encloses any area they 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()**

NXUnnameObject() → See NXGetNamedObject()

NXUpdateDynamicServices()

SUMMARY Re-register provided services

DECLARED IN appkit/Listener.h

SYNOPSIS void **NXUpdateDynamicServices**(void)

DESCRIPTION **NXUpdateDynamicServices()** is used by a service-providing application to re-register the services it's willing to provide. To do this, you create a file with the extension ".service" and place it in the application's path, or in **/NextLibrary/Services**, **/LocalLibrary/Services**, or **~/Library/Services**. The content of the file is identical to a normal service file (see the "Other Features" section for a description of service file format). You then call this function.

NXUserAborted(), NXResetUserAbort()

SUMMARY Report user's request to abort

DECLARED IN appkit/Application.h

SYNOPSIS 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()

NXWindowList() → See NXCountWindows()

NXWriteColor() → See NXReadColor()

NXWriteColorToPasteboard() → See NXReadColorFromPasteboard()

NXWritePoint() → See NXReadPoint()

NXWriteRect() → See NXReadPoint()

NXWriteSize() → See NXReadPoint()

NXWriteWordTable() → See NXReadWordTable()

NXYellowComponent() → See NXRedComponent()

NX_ASSERT()

SUMMARY Write an error message

DECLARED IN appkit/nextstd.h

SYNOPSIS 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_FREE() → See NX_MALLOC()

NX_HEIGHT() → See NX_X()

NX_MALLOC(), NX_REALLOC(), NX_FREE()

SUMMARY Allocate memory

DECLARED IN appkit/nextstd.h

SYNOPSIS
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-name* 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-name*. **NX_REALLOC()** should be used to change the size of the object *var*, just as **realloc()** would be used. 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((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((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

DECLARED IN appkit/nextstd.h

SYNOPSIS 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 does nothing if the application is compiled with NX_BLOCKPSDEBUG defined.

NX_REALLOC() → See NX_MALLOC()

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

DECLARED IN appkit/graphics.h

SYNOPSIS
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

DECLARED IN appkit/nextstd.h

SYNOPSIS *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 in Chapter 3.

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. 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.

NX_ZONEREALLOC() → See NX_ZONEMALLOC()

Types and Constants

Defined Types

NXAcknowledge

DECLARED IN appkit/Listener.h

SYNOPSIS

```
typedef struct _NXAcknowledge {
    msg_header_t header;
    msg_type_t sequenceType;
    int sequence;
    msg_type_t errorType;
    int error;
} NXAcknowledge
```

DESCRIPTION NXAcknowledge is the structure of a Listener acknowledgement message.

NXAppkitErrorTokens

DECLARED IN appkit/errors.h

SYNOPSIS

```
typedef enum _NXAppkitErrorTokens {
    NX_longLine = NX_APPKIT_ERROR_BASE,
    NX_nullSel,
    NX_wordTablesWrite,
    NX_wordTablesRead,
    NX_textBadRead,
    NX_textBadWrite,
    NX_powerOff,
    NX_pasteboardComm,
    NX_mallocError,
    NX_printingComm,
    NX_abortModal,
    NX_abortPrinting,
    NX_illegalSelector,
```

NX_appkitVMError,
NX_badRtfDirective,
NX_badRtfFontTable,
NX_badRtfStyleSheet,
NX_newerTypedStream,
NX_tiffError,
NX_printPackageError,
NX_badRtfColorTable,
NX_journalAborted,
NX_draggingError,
NX_colorUnknown,
NX_colorBadIO,
NX_colorNotEditable,
NX_badBitmapParams,
NX_windowServerComm,
NX_unavailableFont,
NX_PPDIncludeNotFound,
NX_PPDParseError,
NX_PPDIncludeStackOverflow,
NX_PPDIncludeStackUnderflow,
NX_rtfPropOverflow
} **NXAppkitErrorTokens**;

DESCRIPTION This enumeration defines the exceptions raised by the Application Kit. (See **NX_RAISE()** for more information.) The constants are:

NX_longLine	Text class: line longer than 16384 characters
NX_nullSel	Text class: operation attempted on empty selection
NX_wordTablesWrite	Error occurred while writing word tables
NX_wordTablesRead	Error occurred while reading word tables
NX_textBadRead	Text class: error reading from file
NX_textBadWrite	Text class: error writing to file
NX_powerOff	Power off exception
NX_pasteboardComm	Communications problem with pbs server
NX_mallocError	malloc problem
NX_printingComm	Problem sending data to npd
NX_abortModal	abortModal message when not running modal
NX_abortPrinting	Printing aborted
NX_illegalSelector	Invalid selector passed to Application Kit
NX_appkitVMError	Error allocating or deallocating virtual memory
NX_badRtfDirective	Invalid RTF directive
NX_badRtfFontTable	Invalid RTF font table
NX_badRtfStyleSheet	Invalid RTF style sheet

NX_newerTypedStream	Version of typed stream more recent than software
NX_tiffError	Error with TIFF operation
NX_printPackageError	Problem loading the print package
NX_badRtfColorTable	Invalid RTF color table
NX_journalAborted	Journaling session was terminated
NX_draggingError	Error messaging drag service
NX_colorUnknown	NXColorList: unknown color name or number
NX_colorBadIO	NXColorList: file read/write error
NX_colorNotEditable	Attempt to change noneditable color list
NX_badBitmapParams	Inconsistent set of bitmap parameters
NX_windowServerComm	Communications problem with the Window Server
NX_unavailableFont	No default font could be found
NX_PPDIncludeNotFound	Include file in PPD file not found
NX_PPDParseError	PPD parsing error
NX_PPDIncludeStackOverflow	PPD include files nested too deep
NX_PPDIncludeStackUnderflow	PPD include file nesting mismatched
NX_rtfPropOverflow	RTF property stack overflow

NXBreakArray

DECLARED IN appkit/Text.h

SYNOPSIS

```
typedef struct _NXBreakArray {
    NXChunk chunk;
    NXLineDesc breaks[1];
} NXBreakArray;
```

DESCRIPTION An NXBreakArray holds line break information for a Text object. 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.

Since the structure's first field is an NXChunk structure, NXBreakArrays can be manipulated using the functions that manage variable-sized arrays of records. See **NXChunkMalloc()** for more information.

NXCharArray

DECLARED IN appkit/Text.h

SYNOPSIS

```
typedef struct _NXCharArray {
    NXChunk chunk;
    wchar text[1];
} NXCharArray;
```

DESCRIPTION This structure holds holds the character array for the current line in the Text object. Since the structure's first field is an NXChunk structure, NXCharArrays can be manipulated using the functions that manage variable-sized arrays of records. See **NXChunkMalloc()** for more information.

NXCharFilterFunc

DECLARED IN appkit/Text.h

SYNOPSIS

```
typedef unsigned short (*NXCharFilterFunc)
    (unsigned short charCode,
    int flags,
    unsigned short charSet);
```

DESCRIPTION The character filter function analyses each character the user enters in the Text object. See **setCharFilter:** (Text class).

NXCharMetrics

DECLARED IN appkit/afm.h

SYNOPSIS

```
typedef struct {
    short charCode;
    unsigned char numKernPairs;
    unsigned char reserved;
    float xWidth;
    int name;
    float bbox[4];
    int kernPairIndex;
} NXCharMetrics;
```

DESCRIPTION An NXCharMetrics structure stores information on a character. The fields are:

charCode	Character code, −1 if unencoded
numKernPairs	Number of kerning pairs starting with this character
xWidth	Width in x of this character
name	Name—an index into a string table
bbox	Character bounding box
kernPairIndex	Index into **NXFontMetrics.kerns** array

NXChunk

DECLARED IN appkit/chunk.h

SYNOPSIS

```
typedef struct _NXChunk {
    short growby;
    int allocated;
    int used;
} NXChunk;
```

DESCRIPTION NXChunk structures are used to implement variable sized arrays of records. Allocation is by the given size (in bytes)—typically a multiple number of records, say 10. The block of memory never shrinks, and the chunk records the current number of elements. To use NXChunks, declare a structure with an NXChunk structure as its first field. See **NXChunkMalloc()** for more information.

The fields of an NXChunk are:

growby	The increment used to enlarge the array
allocated	How many elements are currently allocated
used	How many elements are currently used

NXColorSpace

DECLARED IN appkit/graphics.h

SYNOPSIS

```
typedef enum _NXColorSpace {
    NX_CustomColorSpace = -1,
    NX_OneIsBlackColorSpace = 0,
    NX_OneIsWhiteColorSpace = 1,
    NX_RGBColorSpace = 2,
    NX_CMYKColorSpace = 5
} NXColorSpace;
```

DESCRIPTION Used to represent sample-encoding formats for a bitmap image.

NXCompositeChar

DECLARED IN appkit/afm.h

SYNOPSIS

```
typedef struct {
    int compCharIndex;
    int numParts;
    int firstPartIndex;
} NXCompositeChar;
```

DESCRIPTION An NXCompositeChar structure describes a composite character. The fields are:

compCharIndex	Index into **NXFontMetrics.charMetrics**
numParts	Number of parts making up this char
firstPartIndex	Index of first part in **NXFontMetrics.compositeCharParts**

NXCompositeCharPart

DECLARED IN appkit/afm.h

SYNOPSIS

```
typedef struct {
    int partIndex;
    float dx;
    float dy;
} NXCompositeCharPart;
```

DESCRIPTION NXCompositeCharPart structures are used to describe elements of a composite character array. The fields are:

partIndex	Index into **NXFontMetrics.charMetrics**
dx	Displacement of part in x
dy	Displacement of part in y

NXDataLinkDisposition

DECLARED IN appkit/NXDataLink.h

SYNOPSIS

```
typedef enum _NXDataLinkDisposition {
    NX_LinkInDestination = 1,
    NX_LinkInSource = 2,
    NX_LinkBroken = 3
} NXDataLinkDisposition
```

DESCRIPTION Returned by NXDataLink's **disposition** method to identify a link as a destination link, a source link, or a broken link. See the NXDataLink class specification for more information on the dispositions of links.

NXDataLinkNumber

DECLARED IN appkit/NXDataLink.h

SYNOPSIS typedef int **NXDataLinkNumber**;

DESCRIPTION The type returned by NXDataLink's **linkNumber** method as a persistent identifier of a destination link.

NXDataLinkUpdateMode

DECLARED IN appkit/NXDataLink.h

SYNOPSIS
```
typedef enum _NXDataLinkUpdateMode {
        NX_UpdateContinuously = 1,
        NX_UpdateWhenSourceSaved = 2,
        NX_UpdateManually = 3,
        NX_UpdateNever = 4
} NXDataLinkUpdateMode
```

DESCRIPTION Used by NXDataLink's **setUpdateMode:** and **updateMode** methods to identify when a link's data is to be updated.

NXDragOperation

DECLARED IN appkit/drag.h

SYNOPSIS

```
typedef enum _NXDragOperation {
        NX_DragOperationNone = 0,
        NX_DragOperationCopy = 1,
        NX_DragOperationLink = 2,
        NX_DragOperationGeneric = 4,
        NX_DragOperationPrivate = 8,
        NX_DragOperationAll = 15
} NXDragOperation;
```

DESCRIPTION The NXDragOperation constants represent the operations that a dragging destination can perform on the data that a dragged image represents. While a dragging session is in progress, the drag operation values returned by the source and destination objects are compared to determine whether the destination object is valid, and to (automatically) set the appearance of the cursor:

- NX_DragOperationNone. The destination won't accept the dragged-image's data; the cursor isn't changed.
- NX_DragOperationCopy. The destination will copy the data; the cursor is changed to the copy cursor.
- NX_DragOperationLink. The destination will create some sort of link, as appropriate for the data; the cursor is changed to the link cursor.
- NX_DragOperationGeneric. The destination will perform a "standard" operation; the cursor is changed to the move cursor.
- NX_DragOperationPrivate. The source and the destination will negotiate for the data, or otherwise send special messages to each other; the cursor isn't changed.
- NX_DragOperationAll. This should only be used by the dragging source as the value of its drag operation mask.

See the NXDraggingDestination protocol for more information.

NXEncodedLigature

DECLARED IN appkit/afm.h

SYNOPSIS

```
typedef struct {
        unsigned char firstChar;
        unsigned char secondChar;
        unsigned char ligatureChar;
} NXEncodedLigature;
```

DESCRIPTION An NXEncodedLigature structure is used for elements of the encoded ligature array. This structure is used only for those ligatures in which all three characters are encoded. The fields are:

firstChar	Character encoding of first character
secondChar	Character encoding of second character
ligatureChar	Character encoding of ligature

NXErrorReporter

DECLARED IN appkit/errors.h

SYNOPSIS typedef void **NXErrorReporter**(NXHandler **errorState*);

DESCRIPTION This is the type for a function that acts as a application's error reporter. See the description of **NXRegisterErrorReporter()** for more information.

NXFaceInfo

DECLARED IN appkit/Font.h

SYNOPSIS

```
typedef struct _NXFaceInfo {
        NXFontMetrics *fontMetrics;
        int flags;
        struct _fontFlags {
                unsigned int usedInDoc:1;
                unsigned int usedInPage:1;
                unsigned int usedInSheet:1;
        } fontFlags;
        struct _NXFaceInfo *nextFInfo;
} NXFaceInfo;
```

DESCRIPTION NXFaceInfo structures store information about a font and its usage. Its fields are:

fontMetrics	Information form the AFM file
flags	Which font information is present
fontFlags	Font usage (see below)
nextFInfo	Pointer to next record in the linked list

The fontFlags substructure records font usage so that conforming PostScript comments can be generated for a document. Its fields are:

usedInDoc	Has the font been used in the document?
usedInPage	Has the font been used in the page?
usedInSheet	Has the font been used in the sheet? (There can be more than one page printed on a sheet of paper.)

NXFontMetrics

DECLARED IN appkit/afm.h

SYNOPSIS

```
typedef struct _NXFontMetrics {
        char *formatVersion;
        char *name;
```

```
    char *fullName;
    char *familyName;
    char *weight;
    float italicAngle;
    char isFixedPitch;
    char isScreenFont;
    short screenFontSize;
    float fontBBox[4];
    float underlinePosition;
    float underlineThickness;
    char *version;
    char *notice;
    char *encodingScheme;
    float capHeight;
    float xHeight;
    float ascender;
    float descender;
    short hasYWidths;
    float *widths;
    unsigned int widthsLength;
    char *strings;
    unsigned int stringsLength;
    char hasXYKerns;
    short *encoding;
    float *yWidths;
    NXCharMetrics *charMetrics;
    int numCharMetrics;
    NXLigature *ligatures;
    int numLigatures;
    NXEncodedLigature *encLigatures;
    int numEncLigatures;
    union {
        NXKernPair *kernPairs;
        NXKernXPair *kernXPairs;
    } kerns;
    int numKernPairs;
    NXTrackKern *trackKerns;
    int numTrackKerns;
    NXCompositeChar *compositeChars;
    int numCompositeChars;
    NXCompositeCharPart *compositeCharParts;
    int numCompositeCharParts;
} NXFontMetrics;
```

DESCRIPTION The NXFontMetrics structure is used to describe a font. (See the description of **readMetrics:** in the Font class specification for more information.)

The structure's fields are:

formatVersion	Version of afm file format
name	Name of font for **findfont**
fullName	Full name of font
familyName	Font family name
weight	Weight of font
italicAngle	Degrees counterclockwise from vertical
isFixedPitch	Is the font monospaced?
isScreenFont	Is the font a screen font?
screenFontSize	If it is, how big is it?
fontBBox[4]	Bounding box (llx, lly, urx, ury)
underlinePosition	Distance from baseline for underlines
underlineThickness	Thickness of underline stroke
version	Version identifier
notice	Trademark or copyright
encodingScheme	Default encoding vector
capHeight	Top of 'H'
xHeight	Top of 'x'
ascender	Top of 'd'
descender	Bottom of 'p'
hasYWidths	Do any chars have non-0 y width?
widths	Character widths in x
widthsLength	
strings	Table of strings and other info
stringsLength	
hasXYKerns	Do any of the kerning pairs have nonzero dy?
encoding	256 offsets into NXCharMetrics
yWidths	Character widths in y (*not* in encoding order, but a parallel array to the NXCharMetrics array)
charMetrics	Array of NXCharMetrics
numCharMetrics	Number of elements
ligatures	Array of NXLigatures
numLigatures	Number of elements
encLigatures	Array of NXEncodedLigatures
numEncLigatures	Number of elements
kerns.kernPairs	Array of NXKernPairs
kerns.kernXPairs	Array of NXKernXPairs
numKernPairs	Number of elements
trackKerns	Array of NXTrackKerns

numTrackKerns	Number of elements
compositeChars	Array of NXCompositeChars
numCompositeChars	Number of elements
compositeCharParts	Array of NXCompositeCharParts
numCompositeCharParts	Number of elements

NXFontTraitMask

DECLARED IN appkit/FontManager.h

SYNOPSIS

```
typedef unsigned int NXFontTraitMask;
```

DESCRIPTION A NXFontTraitMask characterizes one or more of a font's traits. It's used as an argument type for several of the methods in the FontManager class.

NXFSM

DECLARED IN appkit/Text.h

SYNOPSIS

```
typedef struct _NXFSM {
    const struct _NXFSM  *next;
    short delta;
    short token;
} NXFSM;
```

DESCRIPTION NXFSM is a word definition finite-state machine transition structure used by a Text object. The fields are:

next	Points to state to go to; NULL implies final state
delta	If final state, this undoes lookahead
token	If final state, negative value implies word is newline; 0 implies dark; and positive implies white space

NXHeightChange

DECLARED IN appkit/Text.h

SYNOPSIS

```
typedef struct _NXHeightChange {
    NXLineDesc lineDesc;
    NXHeightInfo heightInfo;
} NXHeightChange;
```

DESCRIPTION This structure associates line descriptors and line height information in a Text object.

NXHeightInfo

DECLARED IN appkit/Text.h

SYNOPSIS

```
typedef struct _NXHeightInfo {
    NXCoord newHeight;
    NXCoord oldHeight;
    NXLineDesc lineDesc;
} NXHeightInfo;
```

DESCRIPTION This structure is used to store height information for each line of text in a Text object. The fields are

newHeight	Line height from current position forward
oldHeight	Height before change
lineDesc	Line descriptor

NXJournalHeader

DECLARED IN appkit/NXJournaler.h

SYNOPSIS

```
typedef struct {
    int version;
    unsigned int offsetToAppNames;
    unsigned int lastEventTime;
} NXJournalHeader
```

DESCRIPTION The NXJournalHeader type defines the header for a journaling event file. The event data begins immediately after the header.

NXKernPair

DECLARED IN appkit/afm.h

SYNOPSIS

```
typedef struct {
    int secondCharIndex;
    float dx;
    float dy;
} NXKernPair;
```

DESCRIPTION The NXKernPair structure describes a kerning pair element. Its fields are:

secondCharIndex	Index into NXFontMetrics.charMetrics
dx	x displacement relative to first character
dy	y displacement relative to first character

NXKernXPair

DECLARED IN appkit/afm.h

SYNOPSIS

```
typedef struct {
    int secondCharIndex;
    float dx;
} NXKernXPair;
```

DESCRIPTION The NXKernXPair structure describes a kerning pair element. In this structure, the displacement in the y direction is assumed to be 0. The structure's fields are:

secondCharIndex	Index into NXFontMetrics.charMetrics
dx	X displacement relative to first character

NXLay

DECLARED IN appkit/Text.h

SYNOPSIS

```
typedef struct _NXLay {
    NXCoord x;
    NXCoord y;
    short offset;
    short chars;
    id font;
    void *paraStyle;
    NXRun *run;
    NXLayFlags lFlags;
} NXLay;
```

DESCRIPTION A Text object's NXLay structure represents a single sequence of text in a line and records everything needed to select or draw that piece. The fields are:

x	x coordinate of **moveto**
y	y coordinate of **moveto**
offset	Offset in line array for text
chars	Number of characters in the lay
font	Font object
parastyle	Implementation dependent style sheet information
run	Text run for this lay
lFlags	Lay flags

NXLayArray

DECLARED IN appkit/Text.h

SYNOPSIS
```
typedef struct _NXLayArray {
    NXChunk chunk;
    NXLay lays[1];
} NXLayArray;
```

DESCRIPTION A Text object's NXLayArray structure holds the layout for the current line. Since the structure's first field is an NXChunk structure, NXLayArrays can be manipulated using the functions that manage variable-sized arrays of records. See **NXChunkMalloc()** for more information.

NXLayFlags

DECLARED IN appkit/Text.h

SYNOPSIS
```
typedef struct {
    unsigned int mustMove:1;
    unsigned int isMoveChar:1;
} NXLayFlags;
```

DESCRIPTION This structure records whether a text lay in a Text object needs special treatment. Its fields are:

mustMove	True if current lay follows lay with nonprinting character
isMoveChar	True if lay contains nonprinting character

NXLayInfo

DECLARED IN appkit/Text.h

SYNOPSIS

```
typedef struct _NXLayInfo {
        NXRect rect;
        NXCoord descent;
        NXCoord width;
        NXCoord left;
        NXCoord right;
        NXCoord rightIndent;
        NXLayArray *lays;
        NXWidthArray *widths;
        NXCharArray *chars;
        NXTextCache cache;
        NXRect *textClipRect;
        struct _lFlags {
                unsigned int horizCanGrow:1;
                unsigned int vertCanGrow:1;
                unsigned int erase:1;
                unsigned int ping:1;
                unsigned int endsParagraph:1;
                unsigned int resetCache:1;
        } lFlags;
} NXLayInfo;
```

DESCRIPTION A Text object's NXLayInfo structure is used by the scanning and drawing functions to communicate information about lines. Its fields are:

rect	Bounds rect for current line
descent	Descent line; can be reset by the scanning function
width	Width of line
left	Coordinate visible at left side
right	Coordinate visible at right side
rightIndent	How much white space to leave at right side of line
lays	Filled with NXLay items by the scanning function
widths	Filled with character widths by the scanning function
chars	Filled with characters by the scanning function
cache	Cache of current block and run
textClipRect	If non-nil, the current clipping rectangle for drawing
lFlags.horizCanGrow	1 if the scanning function should dynamically resize x margins
lFlags.vertCanGrow	1 if the scanning function should dynamically resize y margins
lFlags.erase	Tells the drawing function whether to erase before drawing line
lFlags.ping	Tells the drawing function whether to ping the Window Server
lFlags.endsParagraph	True if this line ends the paragraph
lFlags.resetCache	Used in the scanning function to reset local caches

---

NXLigature

DECLARED IN appkit/afm.h

SYNOPSIS
```
typedef struct {
    int firstCharIndex;
    int secondCharIndex;
    int ligatureIndex;
} NXLigature;
```

DESCRIPTION This structure correlates two characters and a ligature character. Its fields are:

firstCharIndex	Index into NXFontMetrics.charMetrics
secondCharIndex	Index into NXFontMetrics.charMetrics
ligatureIndex	Index into NXFontMetrics.charMetrics

NXLineDesc

DECLARED IN appkit/Text.h

SYNOPSIS typedef short **NXLineDesc**;

DESCRIPTION An NXLineDesc is used to identify lines in the Text object.

NXLinkEnumerationState

DECLARED IN appkit/NXDataLinkManager.h

SYNOPSIS
```
typedef struct {
        void *a;
        void *b;
} NXLinkEnumerationState
```

DESCRIPTION An **NXLinkEnumerationState** structure is prepared by NXDataLinkManager's **prepareEnumerationState:** method and then passed to the **nextLinkUsing:** method, allowing an application to retrieve the link manager's links. The contents of this structure are private.

NXMeasurementUnit

DECLARED IN appkit/PageLayout.h

SYNOPSIS
```
typedef enum _NXMeasurementUnit {
        NX_UnitInch,
        NX_UnitCentimeter,
        NX_UnitPoint,
        NX_UnitPica
} NXMeasurementUnit;
```

DESCRIPTION These are the units of measurement that are used by the PageLayout class. They're offered to the user through the Units pop-up list in the Page Layout panel.

NXMessage

DECLARED IN appkit/Listener.h

SYNOPSIS
```
typedef struct _NXMessage {
    msg_header_t header;
    msg_type_t sequenceType;
    int sequence;
    msg_type_t actionType;
    char action[NX_MAXMESSAGE];
} NXMessage
```

DESCRIPTION NXMessage is the structure of messages sent by Speaker objects.

NXModalSession

DECLARED IN appkit/Application.h

SYNOPSIS
```
typedef struct _NXModalSession {
    id app;
    id window;
    struct _NXModalSession *prevSession;
    int oldRunningCount;
    BOOL oldDoesHide;
    BOOL freeMe;
    int winNum;
    NXHandler *errorData;
} NXModalSession;
```

DESCRIPTION The NXModalSession structure contains information used by the system between **beginModalSession:for:** and **endModalSession:** messages. The application should not access any of the fields of this structure.

NXParagraphProp

DECLARED IN appkit/Text.h

SYNOPSIS

```
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;
```

DESCRIPTION These constants are used to identify specific paragraph properties for modification. See Text's **setSelProp:to:** method for more information.

NXParamValue

DECLARED IN appkit/Listener.h

SYNOPSIS

```
typedef union {
        int ival;
        double dval;
        port_t pval;
        struct _bval {
                char *p;
                int len;
        } bval;
} NXParamValue
```

DESCRIPTION Used by Speaker objects to pass method parameters.

NXRect

DECLARED IN appkit/graphics.h

SYNOPSIS

```
typedef struct _NXRect {
        NXPoint origin;
        NXSize size;
} NXRect
```

DESCRIPTION Used throughout the Application Kit to give the dimensions and location of a rectangle on the screen. The NXPoint and NXSize structures are described in Chapter 5, "Display PostScript."

NXRemoteMethod

DECLARED IN appkit/Listener.h

SYNOPSIS

```
typedef struct _NXRemoteMethod {
        SEL key;
        char *types;
} NXRemoteMethod
```

DESCRIPTION Defines a method understood by a Listener.

NXResponse

DECLARED IN appkit/Listener.h

SYNOPSIS

```
typedef struct _NXResponse {
        msg_header_t header;
        msg_type_t sequenceType;
        int sequence;
} NXResponse
```

DESCRIPTION NXResponse is the structure of a Listener response message.

NXRTFDError

DECLARED IN appkit/NXRTFDErrors.h

SYNOPSIS

```
typedef enum {
    NX_RTFDErrorNone
    NX_RTFDErrorSaveAborted,
    NX_RTFDErrorUnableToWriteFile,
    NX_RTFDErrorUnableToCloseFile,
    NX_RTFDErrorUnableToCreatePackage
    NX_RTFDErrorUnableToCreateBackup,
    NX_RTFDErrorUnableToDeleteBackup,
    NX_RTFDErrorUnableToDeleteTemp,
    NX_RTFDErrorUnableToDeleteOriginal,
    NX_RTFDErrorFileDoesntExist,
    NX_RTFDErrorUnableToReadFile,
    NX_RTFDErrorInsufficientAccess,
    NX_RTFDErrorMalformedRTFD
} NXRTFDError;
```

DESCRIPTION This enumeration defines the constants returned by methods that open or save RTFD documents (for example, the **openRTFDFrom:** method in the Text class). These constants divide into four group, as listed in the lists below.

No Errors

NX_RTFDErrorNone

Write Errors

NX_RTFDErrorSaveAborted
NX_RTFDErrorUnableToWriteFile
NX_RTFDErrorUnableToCloseFile
NX_RTFDErrorUnableToCreatePackage
NX_RTFDErrorUnableToCreateBackup
NX_RTFDErrorUnableToDeleteBackup
NX_RTFDErrorUnableToDeleteTemp
NX_RTFDErrorUnableToDeleteOriginal

Read Errors

NX_RTFDErrorFileDoesntExist
NX_RTFDErrorUnableToReadFile

Read/Write Errors

NX_RTFDErrorInsufficientAccess
NX_RTFDErrorMalformedRTFD

NXRun

DECLARED IN appkit/Text.h

SYNOPSIS

```
typedef struct _NXRun {
    id font;
    int chars;
    void *paraStyle;
    float textGray;
    int textRGBColor;
    unsigned char superscript;
    unsigned char subscript;
    id info;
    NXRunFlags rFlags;
} NXRun;
```

DESCRIPTION A Text object's NXRun structure represents a single sequence of text with a given format. The fields are:

font	The Font object for the run
chars	Number of characters in run
paraStyle	Implementation dependent style sheet information
textGray	Gray value of the text
textRGBColor	Text color (negative if not set)
superscript	Superscript in points
subscript	Subscript in points
info	Available for subclasses of Text
rFlags	Indicates underline, etc.

NXRunArray

DECLARED IN appkit/Text.h

SYNOPSIS

```
typedef struct _NXRunArray {
    NXChunk chunk;
    NXRun runs[1];
} NXRunArray;
```

DESCRIPTION A Text object's NXRunArray structure holds the array of text runs. Since the structure's first field is an NXChunk structure, NXRunArrays can be manipulated using the functions that manage variable-sized arrays of records. See **NXChunkMalloc()** for more information.

NXRunFlags

DECLARED IN appkit/Text.h

SYNOPSIS

```
typedef struct {
    unsigned int underline:1;
    unsigned int graphic:1;
} NXRunFlags;
```

DESCRIPTION A Text object's NXRunFlags structure records whether a run contains graphics or is underlined. Its fields are:

underline	True if text is underlined
graphic	True if graphic is present

NXScreen

DECLARED IN appkit/screens.h

SYNOPSIS

```
typedef struct _NXScreen {
    int screenNumber;
    NXRect screenBounds;
    NXWindowDepth depth;
} NXScreen;
```

DESCRIPTION The NXScreen structure represents a screen. Its fields are:

screenNumber	A unique integer that identifies the screen
screenBounds	The screen's area, reckoned in the screen coordinate system
depth	The amount of memory the screen devotes to each pixel

NXSelPt

DECLARED IN appkit/Text.h

SYNOPSIS

```
typedef struct _NXSelPt {
    int cp;
    int line;
    NXCoord x;
    NXCoord y;
    int c1st;
    NXCoord ht;
} NXSelPt;
```

DESCRIPTION A Text object's NXSelPt structure represents one end of a selection. Its fields are:

cp	Character position
line	Offset of LineDesc in break table
x	x coordinate
y	y coordinate
c1st	Character position of first character on the line
ht	Line height

NXSpellCheckMode

DECLARED IN appkit/NXSpellChecker.h

SYNOPSIS

```
typedef enum {
    NX_CheckSpelling,
    NX_CheckSpellingToEnd,
    NX_CheckSpellingFromStart,
    NX_CheckSpellingInSelection,
    NX_CountWords,
    NX_CountWordsToEnd,
    NX_CountWordsInSelection
} NXSpellCheckMode;
```

DESCRIPTION Used as arguments to NXSpellChecker's **checkSpelling:of:** and **checkSpelling:of:wordCount:** methods to specify the extent and nature of word checking and counting. The elements are:

NX_CheckSpelling	Checks spelling of the entire text stream
NX_CheckSpellingToEnd	Checks spelling from the current position to the end
NX_CheckSpellingFromStart	Checks spelling of the stream from top to bottom
NX_CheckSpellingInSelection	Check spelling within the selection
NX_CountWords	Counts the number of words in the entire text stream
NX_CountWordsToEnd	Counts words from the current position to the end
NX_CountWordsInSelection	Counts words in the selection

NXStreamSeekMode

DECLARED IN appkit/readOnlyTextStream.h

SYNOPSIS

```
typedef enum {
    NX_StreamStart,
    NX_StreamCurrent,
    NX_StreamEnd
} NXStreamSeekMode;
```

DESCRIPTION Used by the NXReadOnlyTextStream protocol during a seek on a stream. See the protocol specification for details.

NXStringOrderTable

DECLARED IN appkit/Text.h

SYNOPSIS

```
typedef struct {
    unsigned char primary[256];
    unsigned char secondary[256];
    unsigned char primaryCI[256];
    unsigned char secondaryCI[256];
} NXStringOrderTable;
```

DESCRIPTION The arrays in a Text object's NXStringOrderTable structure are used for case-sensitive and case-insensitive ordering of characters. See the documentation for **NXOrderStrings**() for more information.

NXTabStop

DECLARED IN appkit/Text.h

SYNOPSIS

```
typedef struct _NXTabStop {
    short kind;
    NXCoord x;
} NXTabStop;
```

DESCRIPTION This structure is used to describe a Text object's tab stops. Its fields are:

kind	Kind of tab (only NX_LEFTTAB is currently implemented)
x	x coordinate for stop

NXTextBlock

DECLARED IN appkit/Text.h

SYNOPSIS

```
typedef struct _NXTextBlock {
        struct _NXTextBlock *next;
        struct _NXTextBlock *prior;
        struct _tbFlags {
                unsigned int malloced:1;
        } tbFlags;
        short chars;
        wchar  *text;
} NXTextBlock;
```

DESCRIPTION A Text object's NXTextBlock structures hold the characters of the text. Its fields are:s

next	Next block in linked list
prior	Previous block in linked list
tbFlags.malloced	True if the block was malloc'ed
chars	Number of characters in this block
text	The text in this block

NXTextCache

DECLARED IN appkit/Text.h

SYNOPSIS

```
typedef struct _NXTextCache {
        int curPos;
        NXRun *curRun;
        int runFirstPos;
        NXTextBlock *curBlock;
        int blockFirstPos;
} NXTextCache;
```

DESCRIPTION A Text object's NXTextCache structure describes the current text block and run. Its fields are:

curPos	Current position in text stream
curRun	Current run of text
runFirstPos	Character position of first character in current run
curBlock	Current block of text
blockFirstPos	Character position of first character in current block

---

NXTextFilterFunc

DECLARED IN appkit/Text.h

SYNOPSIS
```
typedef char *(*NXTextFilterFunc)
    (id self,
    unsigned char *insertText,
    int *insertLength,
    int position);
```

DESCRIPTION A Text object's text filter function can be used to implement autoindenting and other features. See Text's **setTextFilter:** method.

NXTextFunc

DECLARED IN appkit/Text.h

SYNOPSIS
```
typedef int (*NXTextFunc)
    (id self,
    NXLayInfo *layInfo);
```

DESCRIPTION This is the type for a Text object's scanning and drawing functions, as set through Text's **setScanFunc:** and **setDrawFunc:** methods.

NXTextStyle

DECLARED IN appkit/Text.h

SYNOPSIS

```
typedef struct _NXTextStyle {
        NXCoord indent1st;
        NXCoord indent2nd;
        NXCoord lineHt;
        NXCoord descentLine;
        short alignment;
        short numTabs;
        NXTabStop *tabs;
} NXTextStyle;
```

DESCRIPTION A Text object's NXTextStyle structure describes the text layout and tab stops. Its fields are:

indent1st	How far the first line of the paragraph is indented
indent2nd	How far the second line is indented
lineHt	Line height
descentLine	Distance to descent line from bottom of line
alignment	Alignment mode
numTabs	Number of tab stops
tabs	Array of tab stops

NXTopLevelErrorHandler

DECLARED IN appkit/errors.h

SYNOPSIS typedef void **NXTopLevelErrorHandler**(NXHandler *_errorState_);

DESCRIPTION This is the type for functions that act as a application's top-level error handler. See the description of **NXDefaultTopLevelErrorHandler()** for more information.

NXTrackingTimer

DECLARED IN appkit/timer.h

SYNOPSIS

```
typedef struct _NXTrackingTimer {
        double delay;
        double period;
        DPSTimedEntry te;
        BOOL freeMe;
        BOOL firstTime;
        NXHandler *errorData;
} NXTrackingTimer;
```

DESCRIPTION Information used by the system between calls to **NXBeginTimer()** and **NXEndTimer()**. All the fields in this structure are private.

NXTrackKern

DECLARED IN appkit/afm.h

SYNOPSIS

```
typedef struct {
        int degree;
        float minPointSize;
        float minKernAmount;
        float maxPointSize;
        float maxKernAmount;
} NXTrackKern;
```

DESCRIPTION This structure records track kerning data. The fields are:

degree	Degree of tightness
minPointSize	Minimum cut-off value
minKernAmount	Kerning amount at minPointSize and below
maxPointSize	Maximum cut-off value
maxKernAmount	Kerning amount at maxPointSize and above

NXWidthArray

DECLARED IN appkit/Text.h

SYNOPSIS

```
typedef struct _NXWidthArray {
        NXChunk chunk;
        NXCoord widths[1];
} NXWidthArray;
```

DESCRIPTION A Text object's NXWidthArray structure holds the character widths for the current line. Since the structure's first field is an NXChunk structure, NXWidthArrays can be manipulated using the functions that manage variable-sized arrays of records. See **NXChunkMalloc()** for more information.

NXWindowDepth

DECLARED IN appkit/graphics.h

SYNOPSIS

```
typedef enum _NXWindowDepth {
        NX_DefaultDepth,
        NX_TwoBitGrayDepth,
        NX_EightBitGrayDepth,
        NX_TwelveBitRGBDepth,
        NX_TwentyFourBitRGBDepth
} NXWindowDepth;
```

DESCRIPTION Encodes the depth, or amount of memory, devoted to a single pixel for a window or screen.

wchar

DECLARED IN appkit/Text.h

SYNOPSIS

```
typedef unsigned char wchar;
```

DESCRIPTION This is the type used for the characters within a Text object.

Symbolic Constants

Bits per Character and Integer

DECLARED IN appkit/nextstd.h

SYNOPSIS
NBITSCHAR
NBITSINT

DESCRIPTION These constants define the number of bits per character and the number of bits per integer, respectively.

Boolean Constants

DECLARED IN appkit/nextstd.h

SYNOPSIS

Constant	Value
TRUE	1
FALSE	0

DESCRIPTION These constants define boolean true and false values.

Box Borders

DECLARED IN appkit/Box.h

SYNOPSIS
NX_NOBORDER
NX_LINE
NX_BEZEL
NX_GROOVE

DESCRIPTION These constants represent the four types of borders that can be drawn around a Box object.

Box Title Positions

DECLARED IN appkit/Box.h

SYNOPSIS
NX_NOTITLE
NX_ABOVETOP
NX_ATTOP
NX_BELOWTOP
NX_ABOVEBOTTOM
NX_ATBOTTOM
NX_BELOWBOTTOM

DESCRIPTION These constants represent the locations where a Box's title can be placed with respect to its border. Thus, for example, NX_ABOVETOP means the title is above the top of the border, NX_ATTOP means the title breaks the top border, and so on.

Button and ButtonCell Highlight/Display Types

DECLARED IN appkit/ButtonCell.h

SYNOPSIS
NX_MOMENTARYPUSH
NX_PUSHONPUSHOFF
NX_TOGGLE
NX_SWITCH
NX_RADIOBUTTON
NX_MOMENTARYCHANGE
NX_ONOFF

DESCRIPTION These constants represent the way Buttons and ButtonCells behave when pressed, and how they display their state. See Button's **setType:** method for more information.

Button and ButtonCell Icon Positions

DECLARED IN appkit/Cell.h

SYNOPSIS
NX_TITLEONLY
NX_ICONONLY
NX_ICONLEFT
NX_ICONRIGHT
NX_ICONBELOW
NX_ICONABOVE
NX_ICONOVERLAPS

DESCRIPTION These constants represent the position of a ButtonCell's icon relative to its title. See Button's **setIconPosition:** method for more information.

Cell and ButtonCell Parameters

DECLARED IN appkit/Cell.h

SYNOPSIS
NX_CELLDISABLED
NX_CELLSTATE
NX_CELLEDITABLE
NX_CELLHIGHLIGHTED

NX_LIGHTBYCONTENTS
NX_LIGHTBYGRAY
NX_LIGHTBYBACKGROUND
NX_ICONISKEYEQUIVALENT
NX_OVERLAPPINGICON
NX_ICONHORIZONTAL
NX_ICONLEFTORBOTTOM
NX_CHANGECONTENTS
NX_BUTTONINSET

DESCRIPTION These constants represent parameters that are accessed through Cell's and ButtonCell's **setParameter:to:** and **getParameter:** methods. Only the first four constants listed above are accessible by Cell; the others apply to ButtonCells only.

Cell Data Entry Types

DECLARED IN appkit/Cell.h

SYNOPSIS
NX_ANYTYPE
NX_INTTYPE
NX_POSINTTYPE
NX_FLOATTYPE
NX_POSFLOATTYPE
NX_DOUBLETYPE
NX_POSDOUBLETYPE

DESCRIPTION These constants represent the numeric data types that a text Cell can accept. See Cell's **setEntryType:** method for more information.

Cell Periodic Action Flag

DECLARED IN appkit/Cell.h

SYNOPSIS NX_PERIODICMASK

DESCRIPTION You pass this constant to Cell's **sendActionOn:** method to indicate that the Cell should send its action message periodically while the mouse is down.

Cell Types

DECLARED IN appkit/Cell.h

SYNOPSIS

Constant	Cell Type
NX_NULLCELL	No display
NX_TEXTCELL	The Cell displays text
NX_ICONCELL	The Cell display an icon

DESCRIPTION These constants represent different types of Cell objects.

Color Panel Modes

DECLARED IN appkit/NXColorPanel.h

SYNOPSIS
NX_GRAYMODE
NX_RGBMODE
NX_CMYKMODE
NX_HSBMODE
NX_CUSTOMPALETTEMODE
NX_CUSTOMCOLORMODE
NX_BEGINMODE

DESCRIPTION These constants represent the different Color panel modes.

Color Panel Mode Masks

DECLARED IN appkit/NXColorPanel.h

SYNOPSIS
NX_GRAYMODEMASK
NX_RGBMODEMASK
NX_CMYKMODEMASK
NX_HSBMODEMASK
NX_CUSTOMPALETTEMODEMASK
NX_LISTMODEMASK
NX_WHEELMODEMASK
NX_ALLMODESMASK

DESCRIPTION These constants provide masks for the Color panel modes.

Color Picker Insertion Order Constants

DECLARED IN appkit/NXColorPanel.h

SYNOPSIS

Insertion Order	Value
NX_WHEEL_INSERTION	0.50
NX_SLIDERS_INSERTION	0.51
NX_CUSTOMPALETTE_INSERTION	0.52
NX_LIST_INSERTION	0.53

DESCRIPTION These constants represent the insertion orders that correspond to the color pickers that are provided by the system.

Drawing Activity States

DECLARED IN appkit/View.h

SYNOPSIS

Constant	Activity
NX_DRAWING	Drawing to the screen
NX_PRINTING	Spooling to a printer
NX_COPYING	Copying to a pasteboard

DESCRIPTION Describes an application's current drawing activity.

Error Base Constants

DECLARED IN appkit/errors.h

SYNOPSIS

NX_APPKIT_ERROR_BASE
NX_APP_ERROR_BASE

DESCRIPTION These constants represent the base error codes for errors generated by the Application Kit and by your application. 1000 error codes are reserved for both sets of errors.

Application Priority Levels

DECLARED IN appkit/Application.h

SYNOPSIS

Level	Value	Meaning
NX_BASETHRESHOLD	1	Normal execution
NX_RUNMODALTHRESHOLD	5	An attention panel is being run
NX_MODALRESPTHRESHOLD	10	A modal event loop is in progress

DESCRIPTION These constants represent the default priorities at which an application runs under the described circumstances. An application's priority setting is used to block the delivery of events that have a lesser priority value. A priority must be between 0 and 30 (inclusive).

Events, Kit-Defined Subtypes

DECLARED IN appkit/Application.h

SYNOPSIS

Constant	Meaning
NX_WINEXPOSED	A nonretained Window has been exposed
NX_APPACT	The application has been activated
NX_APPDEACT	The application has been deactivated
NX_WINMOVED	A Window has moved
NX_SCREENCHANGED	A Window has changed screens

DESCRIPTION These represent events that are manufactured by the Application Kit.

Events, System-Defined Subtype

DECLARED IN appkit/Application.h

SYNOPSIS

Constant	Meaning
NX_POWEROFF	The user is turning off the computer

DESCRIPTION These represent events that are produced by the user's actions on the system.

Figure Space Constant

DECLARED IN appkit/Font.h

SYNOPSIS NX_FIGSPACE

DESCRIPTION This constant identifies the nonbreaking space character in the NeXTSTEP encoding vector.

Font Attribute Constants

DECLARED IN appkit/afm.h

SYNOPSIS
NX_FONTHEADER
NX_FONTMETRICS
NX_FONTWIDTHS
NX_FONTCHARDATA
NX_FONTKERNING
NX_FONTCOMPOSITES

DESCRIPTION The Font class uses these constants to query the Window Server for font attributes. See the description of **readMetrics:** in the Font class specification.

Font Conversion Constants

DECLARED IN appkit/FontManager.h

SYNOPSIS

Type of Change	Value
NX_NOFONTCHANGE	0
NX_VIAPANEL	1
NX_ADDTRAIT	2
NX_SIZEUP	3
NX_SIZEDOWN	4
NX_HEAVIER	5
NX_LIGHTER	6
NX_REMOVETRAIT	7

DESCRIPTION These constants are used as values of a FontManager's **whatToDo** instance variable. The value of this variable determines how the FontManager will convert a font when it receives a **convertFont:** message. (See the description of the FontManager's **convertFont:** method for more information.)

Font Matrix Constants

DECLARED IN appkit/Font.h

SYNOPSIS

NX_IDENTITYMATRIX
NX_FLIPPEDMATRIX

DESCRIPTION These constants identify the orientation of the font. NX_IDENTITYMATRIX identifies a font matrix that's used for fonts that will be displayed in a View having an unflipped coordinate system. If the View has a flipped coordinate system (as is found in a Text object), use NX_FLIPPEDMATRIX.

Font Trait Constants

DECLARED IN appkit/FontManager.h

SYNOPSIS

```
NX_ITALIC
NX_BOLD
NX_UNBOLD
NX_NONSTANDARDCHARSET
NX_NARROW
NX_EXPANDED
NX_CONDENSED
NX_SMALLCAPS
NX_POSTER
NX_COMPRESSED
```

DESCRIPTION These constants are used by the FontManager to identify font traits. The list of font traits should be kept small since the more traits that are assigned to a given font, the harder it will be to map it to some other family. Some traits are mutually exclusive, such as NX_EXPANDED and NX_CONDENSED.

FontPanel View Tags

DECLARED IN appkit/FontPanel.h

SYNOPSIS

```
NX_FPPREVIEWFIELD
NX_FPSIZEFIELD
NX_FPREVERTBUTTON
NX_FPPREVIEWBUTTON
NX_FPSETBUTTON
NX_FPSIZETITLE
NX_FPCURRENTFIELD
```

These tags identify the View objects within a FontPanel object.

Gray Shades

DECLARED IN appkit/graphics.h

SYNOPSIS

Gray Shade	Value
NX_WHITE	1.0
NX_LTGRAY	2.0/3.0
NX_DKGRAY	1.0/3.0
NX_BLACK	0.0

DESCRIPTION These constants represent the four pure (undithered) shades of gray that can be displayed on a monochrome screen.

Icon and Token Window Dimensions

DECLARED IN appkit/Window.h

SYNOPSIS

Dimension	Value
NX_ICONWIDTH	48.0
NX_ICONHEIGHT	48.0
NX_TOKENWIDTH	64.0
NX_TOKENHEIGHT	64.0

DESCRIPTION These constants give the dimensions of an icon and the Window (a token-style Window) in which it's contained.

Image Representation Device Matching Constant

DECLARED IN appkit/NXImageRep.h

SYNOPSIS NX_MATCHESDEVICE

DESCRIPTION This constant is used by NXImageRep to indicate that the value of certain attributes, such as the number of colors, or bits-per-sample, will change to match the device that the image is shown on. See the NXImageRep class specification for more information.

Journaling Flag and Mask

DECLARED IN appkit/Application.h

SYNOPSIS NX_JOURNALFLAG
NX_JOURNALFLAGMASK

DESCRIPTION The flag and associated mask for setting a Window's event mask for journal events.

Journaling Listener Name

DECLARED IN appkit/NXJournaler.h

SYNOPSIS

Name	Value
NX_JOURNALREQUEST	"NXJournalerRequest"

DESCRIPTION This is the name that an Application's master journaler's Listener uses to check into the Network Name Server.

Journaling Recording Device

DECLARED IN appkit/NXJournaler.h

SYNOPSIS
NX_CODEC
NX_DSP

DESCRIPTION Used to set or return the recording device for NXJournaler's **recordDevice** and **setRecordDevice:** methods.

Journaling Status

DECLARED IN appkit/NXJournaler.h

SYNOPSIS
NX_STOPPED
NX_PLAYING
NX_RECORDING
NX_NONABORTABLEFLAG
NX_NONABORTABLEMASK

DESCRIPTION NX_STOPPED, NX_PLAYING, and NX_RECORDING are values of event status and sound status for NXJournaler's **getEventStatus:...** and **setEventStatus:...** methods. If you logically OR NX_NONABORTABLEMASK into the event status for a **setEventStatus:...** message, journaling will be made non-abortable.

Journaling Subevents

DECLARED IN appkit/NXJournaler.h

SYNOPSIS
NX_WINDRAGGED
NX_MOUSELOCATION
NX_LASTJRNEVENT

DESCRIPTION Subevents of the NX_JOURNALEVENT event.

Journaling Window Encodings

DECLARED IN appkit/NXJournaler.h

SYNOPSIS

Encoding	Value
NX_KEYWINDOW	
NX_MAINWINDOW	
NX_MAINMENU	
NX_MOUSEDOWNWINDOW	
NX_APPICONWINDOW	
NX_UNKNOWNWINDOW	

DESCRIPTION Window encodings in ".evt" file used to save journaling sessions.

Listener Maximum Message Size

DECLARED IN appkit/Listener.h

SYNOPSIS NX_MAXMESSAGE

DESCRIPTION The maximum size of a Speaker/Listener remote message.

Listener Maximum Parameters

DECLARED IN appkit/Listener.h

SYNOPSIS NX_MAXMSGPARAMS

DESCRIPTION The maximum number of remote method parameters allowed in a Speaker/Listener remote message. Currently, the maximum is 20.

Listener Position Types

DECLARED IN appkit/Listener.h

SYNOPSIS

Position Type	Value
NX_TEXTPOSTYPE	0
NX_REGEXPRPOSTYPE	1
NX_LINENUMPOSTYPE	2
NX_CHARNUMPOSTYPE	3
NX_APPPOSTYPE	4

DESCRIPTION These constants describe the acceptible values for the *posType* argument in the **msgPosition:posType:ok:** and **msgSetPosition:posType:andSelect:ok:** Speaker/Listener methods.

Listener Reserved Message Numbers

DECLARED IN appkit/Listener.h

SYNOPSIS

Message	Value
NX_SELECTORPMSG	35555
NX_SELECTORFMSG	35556
NX_RESPONSEMSG	35557
NX_ACKNOWLEDGE	35558

DESCRIPTION Reserved values for the **msg_id** field in the **header** field of a Listener's **NXMessage** structure. In other words, these are reserved message numbers for the Mach messages received by a Listener.

Listener RPC Error Return Values

DECLARED IN appkit/Listener.h

SYNOPSIS

Error	Value
NX_INCORRECTMESSAGE	

DESCRIPTION This value is the return value for a Speaker/Listener message that is successfully sent if the selector isn't recognized on the remote side.

Listener Timeout Default

DECLARED IN appkit/Listener.h

SYNOPSIS

Number	Value
NX_SENDTIMEOUT	10000
NX_RCVTIMEOUT	10000

DESCRIPTION These values nominally represent the default timeout values for Speaker/Listener remote messages. However, they are generally disregarded for more reasonable values.

Mach Executable File Segment Names for Images

DECLARED IN appkit/NXImageRep.h

SYNOPSIS

Constant	Segment Name
NX_EPSSEGMENT	"__EPS"
NX_TIFFSEGMENT	"__TIFF"
NX_ICONSEGMENT	"__ICON"

DESCRIPTION These constants represent the three Mach segments in which images can reside.

Matrix Selection Mode Constants

DECLARED IN appkit/Matrix.h

SYNOPSIS
NX_RADIOMODE
NX_HIGHLIGHTMODE
NX_LISTMODE
NX_TRACKMODE

DESCRIPTION These constants represent the modes of operation of a Matrix, as described in the Matrix class specification.

Modal Session Return Values

DECLARED IN appkit/Application.h

SYNOPSIS
NX_RUNSTOPPED
NX_RUNABORTED
NX_RUNCONTINUES

DESCRIPTION Return values for Application's **runModalFor:** and **runModalSession:**.

Open Panel Tag Constants

DECLARED IN appkit/OpenPanel.h

SYNOPSIS
NX_OPICONBUTTON
NX_OPTITLEFIELD
NX_OPCANCELBUTTON
NX_OPOKBUTTON
NX_OPFORM

DESCRIPTION These constants redefine the SavePanel tag constants for the OpenPanel.

Page Layout Panel Button Tags

DECLARED IN appkit/PageLayout.h

SYNOPSIS
NX_PLICONBUTTON
NX_PLTITLEFIELD
NX_PLPAPERSIZEBUTTON
NX_PLLAYOUTBUTTON
NX_PLUNITSBUTTON
NX_PLWIDTHFORM
NX_PLHEIGHTFORM
NX_PLPORTLANDMATRIX
NX_PLSCALEFIELD
NX_PLCANCELBUTTON
NX_PLOKBUTTON

DESCRIPTION These constants represent the tag values of the various buttons that the Page Layout panel displays.

Page Order Modes

DECLARED IN appkit/PrintInfo.h

SYNOPSIS
NX_DESCENDINGORDER
NX_SPECIALORDER
NX_ASCENDINGORDER
NX_UNKNOWNORDER

DESCRIPTION These constants describe the order in which pages are spooled for printing.

Page Orientation Constants

DECLARED IN appkit/PrintInfo.h

SYNOPSIS
NX_PORTRAIT
NX_LANDSCAPE

DESCRIPTION These constants represent the way a page is oriented for printing. In NX_PORTRAIT mode, the page is turned so it's higher than it is wide; NX_LANDSCAPE orients the page to be wider than high.

Pagination Modes

DECLARED IN appkit/PrintInfo.h

SYNOPSIS
NX_AUTOPAGINATION
NX_FITPAGINATION
NX_CLIPPAGINATION

DESCRIPTION These constants represent the different ways in which an image is divided into pages. See the PrintInfo class specification for a fuller explanation.

Panel Button Tags

DECLARED IN appkit/Panel.h

SYNOPSIS

Name	Value
NX_OKTAG	1
NX_CANCELTAG	0

DESCRIPTION These constants define tags for the two buttons commonly presented by a Panel.

Panel Return Values

DECLARED IN appkit/Panel.h

SYNOPSIS

Name	Value
NX_ALERTDEFAULT	1
NX_ALERTALTERNATE	0
NX_ALERTOTHER	–1
NX_ALERTERROR	–2

DESCRIPTION These constants define values returned by the **NXRunAlertPanel()** function and by **runModalSession:** when the modal session is run with a Panel provided by **NXGetAlertPanel()**.

Printer Table Key Length

DECLARED IN appkit/NXPrinter.h

SYNOPSIS NX_PRINTKEYMAXLEN

DESCRIPTION This constant gives the maximum length of a string passed as the key to an NXPrinter printer-information table.

Printer Table States

DECLARED IN appkit/NXPrinter.h

SYNOPSIS

NX_PRINTERTABLEOK
NX_PRINTERTABLENOTFOUND
NX_PRINTERTABLEERROR

DESCRIPTION These constants are used to describe the state of a printer-information table stored by an NXPrinter object.

Rectangle Sides

DECLARED IN appkit/graphics.h

SYNOPSIS

Side	Meaning
NX_XMIN	Parallel to the y-axis, along the side with the smallest x values
NX_YMIN	Parallel to the x-axis, along the side with the smallest y values
NX_XMAX	Parallel to the y-axis, along the side with the greatest x values
NX_YMAX	Parallel to the x-axis, along the side with the greatest y values

DESCRIPTION These constants represent the four sides of a rectangle.

Save Panel Tag Constants

DECLARED IN appkit/SavePanel.h

SYNOPSIS

Name	Value
NX_SPICONBUTTON	150
NX_SPTITLEFIELD	151
NX_SPBROWSER	152
NX_SPCANCELBUTTON	NX_CANCELTAG
NX_SPOKBUTTON	NX_OKTAG
NX_SPFORM	155

DESCRIPTION These constants define tags for identifying views in the SavePanel.

Scroller Arrow Positions

DECLARED IN appkit/Scroller.h

SYNOPSIS

Position	Value
NX_SCROLLARROWSMAXEND	0
NX_SCROLLARROWSMINEND	1
NX_SCROLLARROWSNONE	2

DESCRIPTION These constants are used in Scroller's **setArrowsPosition:** method to set the position of the arrows within the scroller.

Scroller Part Identification Constants

DECLARED IN appkit/Scroller.h

SYNOPSIS

Part	Value
NX_NOPART	0
NX_DECPAGE	1
NX_KNOB	2
NX_INCPAGE	3
NX_DECLINE	4
NX_INCLINE	5
NX_KNOBSLOT	6
NX_JUMP	6

DESCRIPTION These constants are used in Scroller's **hitPart** method to identify the part of the Scroller specified in a mouse event.

Scroller Usable Parts

DECLARED IN appkit/Scroller.h

SYNOPSIS

Usable Parts	Value
NX_SCROLLERNOPARTS	0
NX_SCROLLERONLYARROWS	1
NX_SCROLLERALLPARTS	2

DESCRIPTION These constants define the usable parts of a Scroller object; see the class specification for more information.

Scroller Width and Height

DECLARED IN appkit/Scroller.h

SYNOPSIS NX_SCROLLERWIDTH

DESCRIPTION This constant identifies the default width of a vertical Scroller and the default height of a horizontal Scroller. Currently, the constant is defined as 18.0.

Text Alignment Modes

DECLARED IN appkit/Text.h

SYNOPSIS

NX_LEFTALIGNED
NX_RIGHTALIGNED
NX_CENTERED
NX_JUSTIFIED

DESCRIPTION Used as arguments and return values for methods that specify text alignment.

Text Block Constant

DECLARED IN appkit/Text.h

SYNOPSIS NX_TEXTPER

DESCRIPTION This constant identifies the number of characters to allocate for each text block in a Text object.

Text Key Constants

DECLARED IN appkit/Text.h

SYNOPSIS
NX_BACKSPACE
NX_CR
NX_DELETE
NX_BTAB
NX_ILLEGAL
NX_RETURN
NX_TAB
NX_BACKTAB
NX_LEFT
NX_RIGHT
NX_UP
NX_DOWN

DESCRIPTION These constants are used by a Text object's character filter function.

Text Tab Stop Constant

DECLARED IN appkit/Text.h

SYNOPSIS NX_LEFTTAB

DESCRIPTION This constant identifies the only type of tab currently defined for a Text object.

TIFF Compression Schemes

DECLARED IN appkit/tiff.h

SYNOPSIS
NX_TIFF_COMPRESSION_NONE
NX_TIFF_COMPRESSION_CCITTFAX3
NX_TIFF_COMPRESSION_CCITTFAX4
NX_TIFF_COMPRESSION_LZW
NX_TIFF_COMPRESSION_JPEG
NX_TIFF_COMPRESSION_PACKBITS

DESCRIPTION These constants represent the various TIFF (*tag image file format*) data compression schemes. See the NXBitmapImageRep class specification for their meanings.

View Autoresize Constants

DECLARED IN appkit/View.h

SYNOPSIS
NX_NOTSIZABLE
NX_MINXMARGINSIZABLE
NX_WIDTHSIZABLE
NX_MAXXMARGINSIZABLE
NX_MINYMARGINSIZABLE
NX_HEIGHTSIZABLE
NX_MAXYMARGINSIZABLE

DESCRIPTION Used to describe which parts of a View (or its margins) are resized when the View's superview is resized. See the View class specification for details.

Window Button Masks

DECLARED IN appkit/Window.h

SYNOPSIS
NX_CLOSEBUTTONMASK
NX_MINIATURIZEBUTTONMASK

DESCRIPTION These determine the existence of the close button and miniaturize button in a Window's title bar. See the Window class description for more information.

Window Frame Description String Length

DECLARED IN appkit/Window.h

SYNOPSIS NX_MAXFRAMESTRINGLENGTH

DESCRIPTION You use this constant to allocate a string that will contain Window frame information, as used by Window methods such as **saveFromToString:**.

Window Styles

DECLARED IN appkit/Window.h

SYNOPSIS
NX_PLAINSTYLE
NX_TITLEDSTYLE
NX_MENUSTYLE
NX_MINIWINDOWSTYLE
NX_MINIWORLDSTYLE
NX_TOKENSTYLE
NX_RESIZEBARSTYLE
NX_FIRSTWINSTYLE
NX_LASTWINSTYLE
NX_NUMWINSTYLES

DESCRIPTION Used to describe a Window object's style. The last three constants are useful for sequencing through the list of distinct styles. See the Window class description for more information.

Window Tiers

DECLARED IN appkit/Window.h

SYNOPSIS

Window tier	Value
NX_NORMALLEVEL	0
NX_FLOATINGLEVEL	3
NX_DOCKLEVEL	5
NX_SUBMENULEVEL	10
NX_MAINMENULEVEL	20

DESCRIPTION These constants list the window (device) tiers that are used by the Application Kit. Windows are ordered (or "layered") within tiers: The uppermost window in one tier can still be obscured by the lowest window in the next higher tier.

Workspace Name Constants

DECLARED IN appkit/Listener.h

SYNOPSIS
NX_WORKSPACEREQUEST
NX_WORKSPACEREPLY

DESCRIPTION NX_WORKSPACEREQUEST is the name of the Workspace Manager's Listener's port; it isn't defined until an application enters the run loop. NX_WORKSPACEREPLY is private and shouldn't be meddled with.

Workspace Request Constants

DECLARED IN appkit/workspaceRequest.h

SYNOPSIS

File Operation Constant	Value
WSM_MOVE_OPERATION	"move"
WSM_COPY_OPERATION	"copy"
WSM_LINK_OPERATION	"link"
WSM_COMPRESS_OPERATION	"compress"
WSM_DECOMPRESS_OPERATION	"decompress"
WSM_ENCRYPT_OPERATION	"encrypt"
WSM_DECRYPT_OPERATION	"decrypt"
WSM_DESTROY_OPERATION	"destroy"
WSM_RECYCLE_OPERATION	"recycle"
WSM_DUPLICATE_OPERATION	"duplicate"

DESCRIPTION Possible file operation arguments for the **performFileOperation:source:destination:files:options:** method. The object that responds to this method is available from Application's **workspace** method.

Global Variables

Application Object

DECLARED IN appkit/Application.h

SYNOPSIS id **NXApp**;

DESCRIPTION The current application's Application object.

Break Tables

DECLARED IN appkit/Text.h

SYNOPSIS
const NXFSM *const **NXEnglishBreakTable**;
const int **NXEnglishBreakTableSize**;
const NXFSM *const **NXEnglishNoBreakTable**;
const int **NXEnglishNoBreakTableSize**;
const NXFSM *const **NXCBreakTable**;
const int **NXCBreakTableSize**;

DESCRIPTION These tables are finite state machines that determine word wrapping in a Text object.

Character Category Tables

DECLARED IN appkit/Text.h

SYNOPSIS
onst unsigned char *const **NXEnglishCharCatTable**;
const unsigned char *const **NXCCharCatTable**;

DESCRIPTION These tables define the character classes used in a Text object's break and click tables.

Click Tables

DECLARED IN appkit/Text.h

SYNOPSIS
const NXFSM *const **NXEnglishClickTable**;
const int **NXEnglishClickTableSize**;
const NXFSM *const **NXCClickTable**;
const int **NXCClickTableSize**;

DESCRIPTION These tables are used by a Text object as finite state machines that determine which characters are selected when the user double clicks.

Domain Name

DECLARED IN appkit/Application.h

SYNOPSIS
char *const **NXSystemDomainName**;

DESCRIPTION The name of the host's domain.

File Information

DECLARED IN appkit/workspaceRequest.h

SYNOPSIS
NXAtom **NXPlainFileType**;
NXAtom **NXDirectoryFileType**;
NXAtom **NXApplicationFileType**;
NXAtom **NXFilesystemFileType**;
NXAtom **NXShellCommandFileType**;

DESCRIPTION Values identifying a file's type using the **getInfoForFile:application:type:** method. The object that responds to this message is available from Application's **workspace** method.

File-Name Extension for Data Links

DECLARED IN appkit/NXDataLink.h

SYNOPSIS NXAtom **NXDataLinkFilenameExtension**;

DESCRIPTION The file-name suffix used for links saved to files using NXDataLink's **NXDataLinkFilenameExtension** method.

Null Object

DECLARED IN appkit/Application.h

SYNOPSIS int **NXNullObject**;

DESCRIPTION A canonical null object.

Pasteboard Names

DECLARED IN appkit/Pasteboard.h

SYNOPSIS
NXAtom **NXGeneralPboard**;
NXAtom **NXFontPboard**;
NXAtom **NXRulerPboard**;
NXAtom **NXFindPboard**;
NXAtom **NXDragPboard**;

DESCRIPTION The names of the standard pasteboards. See the Pasteboard class specification introduction for more information.

Pasteboard Types

DECLARED IN appkit/Pasteboard.h

SYNOPSIS
NXAtom **NXAsciiPboardType**;
NXAtom **NXPostScriptPboardType**;
NXAtom **NXTIFFPboardType**;
NXAtom **NXRTFPboardType**;
NXAtom **NXFilenamePboardType**;
NXAtom **NXTabularTextPboardType**;
NXAtom **NXFontPboardType**;
NXAtom **NXRulerPboardType**;
NXAtom **NXFileContentsPboardType**;
NXAtom **NXColorPboardType**;

DESCRIPTION Some standard pasteboard data types. See the Pasteboard class specification for more information.

Pasteboard Types

DECLARED IN appkit/NXDataLink.h

SYNOPSIS
NXAtom **NXDataLinkPboardType**;

DESCRIPTION A pasteboard type for copying a data link to the pasteboard. See the NXDataLink class specification for more information.

Pasteboard Types

DECLARED IN appkit/NXSelection.h

SYNOPSIS NXAtom **NXSelectionPboardType**;

DESCRIPTION A pasteboard type for copying selection descriptions to the pasteboard. See the NXSelection class specification for more information.

Process

DECLARED IN appkit/Application.h

SYNOPSIS int **NXProcessID**;

DESCRIPTION The Mach process in which the current application is running.

Screen Dump Switch

DECLARED IN appkit/View.h

SYNOPSIS BOOL **NXScreenDump**;

DESCRIPTION If YES, objects are printed as they appear on the screen. If NO (the default), objects are printed in their default states.

Smart Cut and Paste Tables

DECLARED IN appkit/Text.h

SYNOPSIS
const unsigned char *const **NXEnglishSmartLeftChars**;
const unsigned char *const **NXEnglishSmartRightChars**;
const unsigned char *const **NXCSmartLeftChars**;
const unsigned char *const **NXCSmartRightChars**;

DESCRIPTION These arrays are suitable as arguments for a Text object's **setPreSelSmartTable:** and **setPostSelSmartTable:** methods. When the user pastes text into a Text object, 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.

View Drawing Status

DECLARED IN appkit/View.h

SYNOPSIS short **NXDrawingStatus**;

DESCRIPTION Encodes the current drawing status for an application. It takes one of the three values listed under "Drawing Activity States," above.

Workspace Name

DECLARED IN appkit/Listener.h

SYNOPSIS
const char ***NXWorkspaceName**;
const char *const **NXWorkspaceReplyName**;

DESCRIPTION Use the Workspace name constants (listed under "Symbolic Constants") rather than these variables.

Other Features

Services

The NeXTSTEP services facility allows an application to make use of the services of other applications without knowing in advance what those services might be. For example, a text editing application lets the system know that it's willing to provide plain ASCII text or rich text (RTF) on the pasteboard any time there is a selection. Any service-providing application is then able to receive that text and act upon it. A service-providing application could thus provide such services as spell checking, grammar checking, encryption, reformatting, language translation, conversion to speech, or any number of useful functions. Service-providing applications can also place data back on the pasteboard to be received by the main application. In this way, data can be seamlessly exchanged between applications, and any application can extend the functionality of many others. This document provides a basic overview of the process of providing and using services.

Providing a Service

In order to provide a service, an application must make known the data types it's willing to act upon, the messages it must receive to initiate action, the menu item to be placed in applications that can provide or accept such data, and the Mach port on which it can receive the messages it published.

As an example, consider a service to reverse text. This service will accept ASCII text on the pasteboard, reverse it, and place the reversed ASCII text back on the pasteboard. Since the Text class supplied with NeXTSTEP knows how place and receive text on the pasteboard, all NeXTSTEP applications will be able to take advantage of the text reversal service to check palindromes or for simple encryption. Since the text will be automatically replaced in a Text object, it will be as though this feature were built in to every application.

First, you must declare the important aspects of the Reverser service in a text file looking something like this:

```
Message: reverseData
Port: Reverser
Send Type: NXAsciiPboardType
Return Type: NXAsciiPboardType
Menu Item: Reverse It
```

This is known as a service specification. (More than these five fields may be listed. The complete specification is described later.) For this example, we will call this file **services.text**. The **Send Type** field indicates that this service requires that data of NXAsciiPboardType be placed on the pasteboard. NXAsciiPboardType is a data type consisting of simple ASCII text. NeXTSTEP defines data types for simple and rich text, file names, encapsulated PostScript, TIFF image data and others. A service is free to

request any data types it likes, including proprietary formats, but the service will only be enabled if the main application can supply data of that type. The **Return Type** field indicates that the service will return ASCII data on the pasteboard after manipulating the data. A return type isn't necessary; the service could simply act on the data it receives without returning anything to the main application. However, since this service returns data, the main application will wait for the service to provide data before it continues processing. (If the service doesn't return data, the service is invoked asynchronously and the main application doesn't wait.) Both the **Send Type** and **Return Type** fields are optional. A service may just accept data, it may just provide data, or it may modify data to be pasted back into the main application. A service may also list multiple send or return types indicating that it can accept any one of several types or that it will return many types; to indicate this, the **Send Type** or **Return Type** lines can be duplicated.

The **Menu Item** field indicates that a **Reverse It** command should be added to the Services menu of every application that can (at least under some circumstances) send and receive ASCII text. This command will be enabled any time a text field has a selection that can be reversed. If the user chooses the enabled **Reverse It** command, the selection will be placed on the pasteboard and the message indicated in the **Message** field will be send to the port indicated in the **Port** field.

How a Service Is Advertised

NeXTSTEP uses the Mach-O executable file format, which effectively provides a simple directory structure to executable files. An executable thus contains multiple segments (akin to directories) each with a number of sections containing binary code, images, text, and other data. At run time, the system will look into the executable files in **~/Apps** and **/LocalApps**. If an executable contains a **__services** section in its **__ICON** segment, the services listed in the section will be made available to the appropriate applications. The following line must be added to the **Makefile.preamble** file (included by NeXTSTEP's standard makefile) to include the above **services.text** file:

```
LDFLAGS = -sectcreate __ICON __services services.text
```

How to Implement a Service

The message line in the **services.text** file indicates that the system will send a **reverseData** message to the service-providing application when the user clicks its menu item. Before sending such a message, the system will ask the main application to put the selected data on the pasteboard in the format required by the service. (The service's menu item will only be enabled when the main application has confirmed that it will be able to provide the data; more on this later.) The actual message sent to the service provider contains parameters identifying the pasteboard, supplying optional information about which service is actually to be performed (since a single method can be used to perform multiple services), and a

pointer allowing the service to return an error message. Here is a possible implementation of the service to reverse text:

```
- reverseData: (id)pasteboard
    userData:(const char *)userData
    error:(char **)msg
{
    const char *types[1];
    char *buffer, *revBuffer, *data;
    int length, i=0, j;

    [pasteboard types];    // pretend to check the pasteboard types

    // read the ASCII data from the pasteboard
    if ([pasteboard readType:NXAsciiPboardType data:&data
        length:&length])
    {
        buffer = malloc(length+1)
        revBuffer = malloc(length+1)

        strncpy(buffer,data,length);
        buffer[length]='\0';
        revBuffer[length]='\0';

        // Reverse the text into revBuffer
        j = length - 1;
        while (j >=0) revBuffer[i++] = buffer[j--];

        // Write the reversed buffer back to the pasteboard
        types[0] = NXAsciiPboardType;
        [pasteboard declareTypes: types num: 1 owner:nil];
        [pasteboard writeType: NXAsciiPboardType data:revBuffer
            length:length];

        free(buffer);
        free(revBuffer);

    }
    else *msg = "Error: couldn't reverse text.";

    return self;
}
```

Every application has a Listener object to receive Objective C messages from external applications. This Listener registers its Mach port with the network name server under the application's name (Reverser, in this case), and it's used to receive messages from the

Workspace Manager to open files or to receive notification that the system will shut down. This same Listener can also be used to receive messages from applications requesting services.

The Listener must be told which object within the application implements the methods that respond to service requests. This object is referred to as the *services delegate*. For example, to inform the application's Listener object that *theServiceObject* is the services delegate, you'd send these messages:

```
id theListener = [NXApp appListener];
[theListener setServicesDelegate:theServiceObject];
```

Thereafter, the Reverser application will receive the **reverseData:userData:error:** message any time the user requests that the selected text be reversed.

Fields in a Service Specification

The following fields must be included in a service specification:

Message: *<name of message>*

This field identifies the method that will be invoked in the Listener's service delegate. In the example above, the message field is **reverseData**, so the delegate must implement a **reverseData:userData:error:** method.

Port: *<name of a port>*

The name of a port of a Listener that is listening for service messages. Since every NeXTSTEP application has, by default, a Listener port registered under the application's name, the sevice specification generally uses this port name.

Menu Item: *<string appearing in other app's Services Menu>*

The menu item for the service. This string will appear in the Services menus of applications that can take advantage of the service. If this string contains a '/' character, that character will be used as a delimiter to specify a second level in the Service menu hierarchy. For example, a menu item of **Encrypt/Replace** will create a submenu **Encrypt** in the Services menu, with a **Replace** menu item in that submenu. Only one level of hierarchy is supported.

The menu item field must be untranslated. Translated menu items can be achieved by including menu item fields preceded by the appropriate language; for example **French Menu Item**. Alternatively, the supplied string can be used as a key into a ".strings" file (see **NXStringTable()**) called ***Language*.lproj/ServicesMenu.strings** found in the same directory as the executable containing the **__services** section

In addition, the following fields may be included in a service specification:

User Data: *<any arbitrary string>*

The **User Data** field is for the service provider's use and is simply passed along as one of the parameters to the **someMessage:userData:error:** message. This parameter may be useful if multiple services are performed by a single method.

Send Type: *<any valid pasteboard type>*

The **Send Type** field specifies the type of data the requesting application is expected to provide in making the request. You may have more than one **Send Type** field (implying that your request can operate on more than one type of data), but the requesting application is required only to place one of those types into the Pasteboard.

Return Type: *<any valid pasteboard type>*

If the **Return Type** field is specified, then the requesting application will expect you to place some data of that type back into the pasteboard object which you are passed. You may specify any number of return types, but you must place ALL of those types in the pasteboard as part of your implementation of your method (though, of course, you may provide some of them lazily—see the Pasteboard documentation's description of the **provideData:** method). Under normal circumstances, the requestor will use the returned data to replace the selection, though the requestor isn't required to do so.

Executable: *<a full path to an executable file>*

The process which actually services a request need not be a full-fledged application with a user-interface, an icon, and Mach-O segments. The **Executable** field lets you specify the path to the program which should be launched before looking up the port. Note that you must still provide a normal application with a user-interface in whose Mach-O you can put the request information (even if the service is always provided by a lightweight program). This full-fledged application should at the very least give a short description of the provided service(s) as well as any copyright or usage information when the user double-clicks on it from the Workspace.

Timeout: *<some number of milliseconds>*

The **Timeout** field is used to determine how long a request might take to process. The default is 30000 milliseconds. Increasing this time allows time consuming services to be performed before the system assumes there was an error and continues. Decreasing this time for speedy services allows errors to be reported more quickly.

Host: *<the name of a network host>*

The **Host** field lets you specify a specific host on which the service provider should be run. This is done either by requesting the launch of that application from the Workspace Manager running on that host or by using **rsh**(1) to start up the application on the remote host (if it isn't a full-fledged application).

Key Equivalent: *<any character>*

The **Key Equivalent** field may be used to specify the key equivalent for the menu item that invokes the service. Like the Menu Item field, it may be localized by preceding it with a language. For example, a service could have the following entries in its service specification:

```
Menu Item: Hello
French Menu Item: Bonjour
Key Equivalent: H
French Key Equivalent: B
```

Specifying Services Dynamically

Many services are known in advance, so the services specification is included in a Mach-O section of the executable file. Some services, however, can't be known until run-time. For example, when data is added to a Librarian bookshelf, Librarian can provide a service to look up information within that data.

To facilitate such dynamic services, you must create a text file with a ".services" extension. Alternatively, you may create a directory with a **.services** extension, and a text file called **services** inside it. The format of this text file is exactly the same as the services specification detailed earlier. This file or directory must be placed in your normal application path or one of **/NextLibrary/Services**, **/LocalLibrary/Services** or **~/Library/Services**. After adding the file, call the function **NXUpdateDynamicServices()** to get the system to recognize your newly-added services.

Using Services

In order to take advantage of services, an application must have a Services menu, and it must contain Responder objects that register the data types that they may be willing to export and import. If the application's interface is generated with Interface Builder, you can simply drag the Services menu item into the application's menu from an Interface Builder palette. If the application's menu is created programmatically, you can specify the menu item that is to be the Services menu with Application's **setServicesMenu:** method.

Registering Types

Responder objects (including subclasses of View, Window, and Application) should, at the time they are created, register all the data types that they can import and export by using Application's **registerServicesMenuSendTypes:andReturnTypes:** method. The lists of types provided to this method need not be balanced; it's perfectly reasonable for a Responder to handle one export type and three import types, for example. Some of the standard pasteboard data types are listed in **appkit/Pasteboard.h**. A Responder doesn't necessarily have to import or export common data types, but more service providers will be able to act on the common data types than on less common types.

The types supplied to **registerServicesMenuSendTypes:andReturnTypes:** are used to determine which service provider commands are listed in the Services menu. Any service provider that can receive a data type provided by the application or that can supply data to the application should be allowed to have an item in the Services menu, so Responders should provide a complete list of the data they use under any circumstance. The item for an individual service provider will be dynamically enabled any time the application can supply or use the data required or supplied by the service.

The following code could be used to register an object that is, at least in some state, able to export ASCII or RTF text and/or import ASCII text:

```
const char *sendTypes[3];
const char *returnTypes[2];
sendTypes[0] = NXAsciiPboardType;
sendTypes[1] = NXRTFPboardType;
sendTypes[2] = NULL;
returnTypes[0] = NXAsciiPboardType;
returnTypes[1] = NULL;
[NXApp registerServicesMenuSendTypes:sendTypes
    andReturnTypes:returnTypes];
```

Validating Services Dynamically

A Responder (or delegate) that can use services must validate the data types that it can import and export at any given time. It does this by implementing the **validRequestorForSendType:andReturnType:** method. This method is invoked for each service that the application might be able to make use of, with arguments for the data types the service requires. If the Responder can, in its current state, use both the specified send and receive data types (or they are **nil**) it should return **self** to indicate that the corresponding service can be enabled. If the responder can't make use of either the send type or the receive type, it should forward the message to its superclass's implementation; the default implementation will then forward the message up the responder chain, looking for a responder that can take advantage of the service.

The **validRequestorForSendType:andReturnType:** method may be invoked frequently, typically many times per event to ensure that the menu items for all service providers reflect the state of the application. A Responder's implementation of this method must be fast so that event handling remains snappy. The arguments to this method are NXAtoms, so you can compare the arguments to standard pasteboard types by comparing pointers rather than comparing strings.

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];
}
```

While the application is running, the **validRequestorForSendType:andReturnType:** message is sent to objects in a limited Responder chain, consisting of the responder chain in the key window, the key window's delegate (only if it isn't a Responder), the Application object, and the Application object's delegate (only if it isn't a Responder). The delegates of the key window and Application object are excluded if they are Responders in order to keep the message from being sent down additional responder chains.

How a Service Is Invoked

A service's menu item is enabled any time the application returns a non-**nil** value to a **validRequestorForSendType:andReturnType:** message. If the user then clicks on the service's menu item, the service is invoked. If the service requires data but doesn't send any back (that is, if the service has a send type but no return type) then the service is invoked asynchronously; the application provides the data and continues to run without waiting on the service. However, if the service provides data (that is, the send type is non-NULL) then the service is invoked synchronously; the application won't continue until the service supplies the data or the service request times out.

When the service is invoked, the system checks whether the service requires data. If so, the responder that returned **self** to the **validRequestorForSendType:andReturnType:** message is sent a **writeSelectionToPasteboard:** message to instruct the responder to provide the data it said it would be able to supply. The implementation of this method should put the data on the pasteboard using the **declareTypes:num:owner:** Pasteboard method. If a pasteboard owner is specified, the responder can wait to provide the actual data by implementing the **pasteboard:provideData:** method. (The owner must persist as long as the application is running.) If no owner is specified, the application should provide the data immediately using Pasteboard's **writeType:data:length:** method.

The responder's implementation of **writeSelectionToPasteboard:** should return YES if the selection is successfully written to the Pasteboard, and NO if it fails to supply the data. However, if the responder correctly replies to **validRequestorForSendType:andReturnType:** queries, it should almost always be able to subsequently provide the data.

If the service returns data (that is, has a non-NULL return type), the application will wait (up to the service's time-out period) for the service to provide the returned data. The service must do its processing work and put the data back on the pasteboard using Pasteboard's **declareTypes:num:owner:** method, as described earlier. The application will then receive a **readSelectionFromPasteboard:** message, and its implementation of that method should replace the selection (which could be empty, like a cursor marking an insertion point) with the data from the pasteboard.

Invoking a Service Programmatically

Though services are usually invoked when the user clicks a service menu item, they may also be invoked programmatically with the following function:

```
BOOL NXPerformService(const char *itemName, Pasteboard *pboard)
```

This function returns YES if the service is successfully performed. *itemName* is a Services menu item in any language. Note that Services menu entries which are in subdirectories

must include a slash wherever there is a subdirectory, for example, "Mail/Selection". The *pboard* must contain whatever data the service requires, and will, upon return of the function, contain the resultant data provided by the service.

Examples of Services

Here are a few examples of services that have already been implemented to give you an idea of what can be done with NeXTSTEP's services mechanism:

- Optical character recognition—When a NeXTSTEP application receives a fax, it receives a bitmap that can't be edited as text. If the application is willing to place the image on the pasteboard as a TIFF image, then an optical character recognition service can convert the image to ASCII text and paste it back as editable data.

- Encryption—An encryption service can convert data to a more secure form. For example, Mail can place a mail message on the pasteboard as a standard Rich Text Format (RTF) document, and another application could encrypt the document and place it back into mail as unreadable ASCII text, or as a document to be opened only by another external decryption application.

- Encapsulated PostScript effects—Many applications support encapsulated PostScript (EPS) graphic images. An EPS effects service can take selected graphics from the pasteboard, rotate them, scale them, and add other effects before pasting them back. In this manner, consistent graphics editing is enabled, even in applications with minimal graphics support.

- Database lookup—Selected topics can be looked up in a database. This is a good example of an asynchronous service that reads data from the pasteboard but doesn't send any data back to the main application.

- Document compression and mailing—One standard pasteboard type defines a complete file name, including its path. Services use this data to send the current file by Mail, and to compress the current document.

- Macro services—Not all services require data from an application. Some simply provide data on request. Examples include macro programs that insert commonly used data such as signatures and time stamps.

3 *Common Classes and Functions*

3 *Common Classes and Functions*

Library: libsys_s.a

Header File Directories: /NextDeveloper/Headers/objc
/NextDeveloper/Headers/streams
/NextDeveloper/Headers/defaults
/NextDeveloper/Headers/appkit

Import: appkit/appkit.h
or individual header files

Introduction

The classes and functions described in this chapter can serve a wide variety of different kinds of applications. They get the name "common" from the fact that they're generally useful and commonly used. Most provide ways of managing data and resources within a program. They can be used in conjunction with any of the NeXTSTEP software kits.

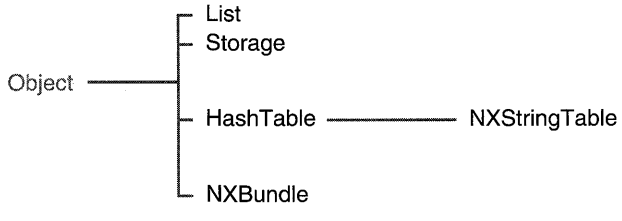


Figure 3-1. Common Classes Inheritance Hierarchy

The table below is a guide to the facilities documented here. The first column states, in a few words, a kind of data-management or resource-management service. The second column says where to look in the chapter; it names the class or principal function that provides the service. Consult the documentation for the functions and classes mentioned to learn whether they're right for your application.

Facility	Class or Function
Storage allocation	NXZoneMalloc() NXCreateZone() NXMallocCheck() Storage class
Unordered collections	HashTable class NXCreateHashTable() NXHashInsert() NXStringTable class
Ordered collections	List class Storage class
Recording user preferences	NXRegisterDefaults()
Localizing resources	NXBundle class NXLocalizedString()
Exception handling	NX_RAISE() NXDefaultExceptionRaiser() NXSetUncaughtExceptionHandler() NXAllocErrorData()
Dynamic loading of code	NXBundle class
Input-output streams	NXOpenFile() NXOpenMemory() NXRead() NXPutc()
Archiving and typed streams	NXOpenTypedStream() NXReadObject() NXReadType()
Classifying and converting characters	NXIsAlpha() NXToAscii()

Because the Application Kit makes use of all the common classes and almost all of the common functions, importing the **appkit.h** header file automatically imports all but one of the header files that declare common classes and functions. Since **appkit.h** corresponds to a precompiled version of the header files, importing it rather than individual files dramatically reduces the time required for compilation. The only common header file that's omitted, and that you might need to import individually, is **streams/streamsimpl.h**. It declares rarely used functions that implement new versions of a stream.

Classes

HashTable

Inherits From: Object

Declared In: objc/HashTable.h

Class Description

The HashTable class defines objects that store associations of keys and values. You use a HashTable object when you need a convenient and efficient way to retrieve the data associated with an arbitrary key. Internally, a hash table locates the key and associated object according to the value returned by applying a *hashing* function to the key. However, the hashing operation is provided automatically by the HashTable's methods, so that the methods that add an association to a HashTable (or return an association, given its key) accept and return the key values directly, not their hashed forms.

In a HashTable object, keys must be of the same type (so that the same hashing function can be applied to each of them), and associated values must be of the same type. The types of the keys and the values are established when the HashTable is initialized. The **initKeyDesc:valueDesc:...** methods take arguments that let you specify key type and value type independently. The **initKeyDesc:** method specifies the type of the keys but assumes that the associated values are **id**s. The **init** method assumes that both keys and associated values are of type **id** (object pointers). The following characters are used as HashTable descriptions (that is, as arguments to the **initKeyDesc:** or **initKeyDesc:valueDesc:** methods):

Character	Type
@	id
*	char *
%	NXAtom
i	int
!	other

Hashing Algorithm and Tests for Equality

The class uses three different algorithms, selected according to the description of the keys. For keys that are of type "object", the HashTable sends itself a **hash** message (inherited from Object). For keys that are strings, it uses a string hashing function. For all other cases, it uses a generic integer hashing function.

To test whether a proposed key is equal to a key already included in the HashTable, keys that are objects are compared using an **isEqual:** message. If two keys are equal in the sense of **isEqual:**, then their hashed values must be equal.

Keys that are strings are compared using a string comparison. Note that the HashTable object keeps only a pointer to the string used as a key (without making a copy of the string), so the string to which it points must never change as long as the association remains in the table.

If you're creating a HashTable whose keys are List or Storage objects, note that these classes have an **isEqual:** method but no **hash** method; you can either subclass or define a **hash** method.

When freeing a HashTable, only object keys or object values are freed.

When a HashTable is archived, data is archived according to its type description. For keys or values whose description is "**%**", upon reading to reconstitute an archived HashTable, each such string is reconstructed by again calling the **NXUniqueString()** function to assure that it is unique in the new context.

Function Interface to Hash Tables

When even greater efficiency of storage and access is required, consider using the C function interface to hash tables rather than the HashTable class (see **NXCreateHashTable**()).

Related Classes

Two other classes for storage and retrieval are NXStringTable and List. An NXStringTable object is a hash table specialized for the situation in which both keys and values are character strings. A List stores a sequential collection of objects; however, it stores the objects (that is, the pointers to them) without keys, so the time required to find a particular element in a List grows linearly with the number of elements.)

Instance Variables

unsigned int **count**;
const char ***keyDesc**;
const char ***valueDesc**;

count	Current number of associations
keyDesc	Description (character representing the type) of keys
valueDesc	Description (character representing the type) of values

Method Types

Initializing and freeing a HashTable	– init – initKeyDesc: – initKeyDesc:valueDesc: – initKeyDesc:valueDesc:capacity: – free – freeObjects – freeKeys:values: – empty
Copying a HashTable	– copyFromZone:
Manipulating table associations	– count – isKey: – valueForKey: – insertKey:value: – removeKey:
Iterating over all associations	– initState – nextState:key:value:
Archiving	– read: – write:

Instance Methods

copyFromZone:

– **copyFromZone:**(NXZone *)*zone*

Returns a new HashTable of the same size as the receiving object. Memory for the new HashTable is allocated from *zone*. Neither keys nor values are copied.

count

– (unsigned int)**count**

Returns the number of objects in the table.

empty

– **empty**

Empties the HashTable but retains its capacity.

free

– **free**

Deallocates the HashTable (but not the objects that its associations point to).

freeKeys:values:

– **freeKeys:**(void (*)(void *))*keyFunc* **values:**(void (*)(void *))*valueFunc*

Conditionally deallocates the HashTable's associations but does not deallocate the table itself.

freeObjects

– **freeObjects**

Deallocates every object in the HashTable, but not the HashTable itself. Strings are not recovered.

init

– **init**

Initializes a new HashTable to map keys of type "object" to values of type "object." Returns **self**.

See also: – **initKeyDesc:key:value:capacity:**

initKeyDesc:

– **initKeyDesc:**(const char *)*aKeyDesc*

Initializes a new HashTable to map keys as described by *aKeyDesc* to object values. Returns **self**.

See also: – **initKeyDesc:key:value:capacity:**

initKeyDesc:valueDesc:

– **initKeyDesc:**(const char *)*aKeyDesc* **valueDesc:**(const char *)*aValueDesc*

Initializes a new HashTable to map keys and values as described by *aKeyDesc* and *aValueDesc*. Returns **self**.

See also: – **initKeyDesc:key:value:capacity:**

initKeyDesc:valueDesc:capacity:

– **initKeyDesc:**(const char *)*aKeyDesc*
valueDesc:(const char *)*aValueDesc*
capacity:(unsigned int)*aCapacity*

Initializes a new HashTable. This is the designated initializer for HashTable objects: If you subclass HashTable, 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.

A HashTable initialized by this method maps keys and values as described by *aKeyDesc* and *aValueDesc*. The argument *aCapacity* is given only as a hint; you can use 0 to create a table of minimal size. As more space is needed, it will be allocated automatically, each time doubling the table's capacity. Returns **self**.

See also: – **initKeyDesc:key:value:capacity:**

initState

– (NXHashState)**initState**

Returns an NXHashState structure that's required when iterating through the HashTable. Iterating through all of a HashTable's associations involves setting up an iteration state, conceptually private to HashTable, and then progressing until all entries have been visited. Here's an example of visiting all the associations in a HashTable called **table** (and just counting them):

```
unsigned int count = 0;
const   void  *key;
        void  *value;
NXHashState  state = [table initState];
while ([table nextState: &state key: &key value: &value])
    count++;
```

See also: – **nextState:key:value:**

insertKey:value:

– (void *)**insertKey:**(const void *)*aKey* **value:**(void *)*aValue*

Adds or updates a key and value pair, as specified by *aKey* and *aValue*. If *aKey* is already in the hash table, it's associated with *aValue* and its previously associated value is returned. Otherwise, **insertKey:value:** returns **nil**.

See also: – **removeKey:**

isKey:

– (BOOL)**isKey:**(const void *)*aKey*

Returns YES if *aKey* is in the table, otherwise NO.

See also: – **valueForKey:**

nextState:key:value:

– (BOOL)**nextState:**(NXHashState *)*aState*
key:(const void **)*aKey*
value:(void **)*aValue*

Moves to the next entry in the HashTable and provides the addresses of pointers to its key/value pair. No **insertKey:** or **removeKey:** should be done while iterating through the table. Returns NO when there are no more entries in the table; otherwise, returns YES. If there are no more entries, *aKey* and *aValue* are set to NULL.

See also: – **initState**

read:

– **read:**(NXTypedStream *)*stream*

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

See also: – **write:**

removeKey:

– (void *)**removeKey:**(const void *)*aKey*

Removes the hash table entry identified by *aKey*. Always returns **nil**.

See also: – **insertKey:value:**

valueForKey:

– (void *)**valueForKey:**(const void *)*aKey*

Returns the value mapped to *aKey*. Returns **nil** if *aKey* is not in the table.

See also: – **isKey:**

write:

– **write:**(NXTypedStream *)*stream*

Writes the HashTable to the typed stream *stream*. Returns **self**.

See also: – **read:**

List

Inherits From: Object

Declared In: objc/List.h

Class Description

A List is a collection of objects. The class provides an interface that permits easy manipulation of the collection as a fixed or variable-sized list, a set, or an ordered collection. Lists are implemented as arrays to allow fast random access using an index. Indices start at 0.

A List array contains object **id**s. An object isn't copied when it's added to a List; only its **id** is. There are no empty slots within the array. **nil** objects can't be inserted in a List, and the collection is contracted to close the empty space when an object is removed.

A List grows dynamically when new objects are added. The default mechanism automatically doubles the capacity of the List when it becomes full, thus ensuring an average constant time for insertions, independent of the size of the List.

For manipulating sets of structures that aren't objects, see the Storage class.

Instance Variables

id ***dataPtr**;
unsigned int **numElements**;
unsigned int **maxElements**;

dataPtr	The data managed by the List object (the array of objects).
numElements	The actual number of objects in the array.
maxElements	The total number of objects that can fit in currently allocated memory.

Method Types

Initializing a new List object	– init – initCount:
Copying and freeing a List	– copyFromZone: – free
Manipulating objects by index	– insertObject:at: – addObject: – removeObjectAt: – removeLastObject – replaceObjectAt:with: – objectAt: – lastObject – count
Manipulating objects by **id**	– addObject: – addObjectIfAbsent: – removeObject: – replaceObject:with: – indexOf:
Comparing and combining Lists	– isEqual: – appendList:
Emptying a List	– empty – freeObjects
Sending messages to the objects	– makeObjectsPerform: – makeObjectsPerform:with:
Managing the storage capacity	– capacity – setAvailableCapacity:
Archiving	– read: – write:

Instance Methods

addObject:

– **addObject:***anObject*

Inserts *anObject* at the end of the List, and returns **self**. However, if *anObject* is **nil**, nothing is inserted and **nil** is returned.

See also: – **insertObject:at:**, – **appendList:**

addObjectIfAbsent:

– **addObjectIfAbsent:***anObject*

Inserts *anObject* at the end of the List and returns **self**, provided that *anObject* isn't already in the List. If *anObject* is in the List, it won't be inserted, but **self** is still returned.

If *anObject* is **nil**, nothing is inserted and **nil** is returned.

See also: – **insertObject:at:**

appendList:

– **appendList:**(List *)*otherList*

Inserts all the objects in *otherList* at the end of the receiving List, and returns **self**. The ordering of the objects is maintained.

See also: – **addObject:**

capacity

– (unsigned int)**capacity**

Returns the maximum number of objects that can be stored in the List without allocating more memory for it. When new memory is allocated, it's taken from the same zone that was specified when the List was created.

See also: – **count**, – **setAvailableCapacity:**

copyFromZone:

– **copyFromZone:**(NXZone *)*zone*

Returns a new List object with the same contents as the receiver. The objects in the List aren't copied; therefore, both Lists contain pointers to the same set of objects. Memory for the new List is allocated from *zone*.

See also: – **copy** (Object)

count

– (unsigned int)**count**

Returns the number of objects currently in the List.

See also: – **capacity**

empty

– **empty**

Empties the List of all its objects without freeing them, and returns **self**. The current capacity of the List isn't changed.

See also: – **freeObjects**

free

– **free**

Deallocates the List object and the memory it allocated for the array of object **id**s. However, the objects themselves aren't freed.

See also: – **freeObjects**

freeObjects

– **freeObjects**

Removes every object from the List, sends each one of them a **free** message, and returns **self**. The List object itself isn't freed and its current capacity isn't altered.

The methods that free the objects shouldn't have the side effect of modifying the List.

See also: – **empty**

indexOf:

– (unsigned int)**indexOf:***anObject*

Returns the index of the first occurrence of *anObject* in the List, or NX_NOT_IN_LIST if *anObject* isn't in the List.

init

– **init**

Initializes the receiver, a new List object, but doesn't allocate any memory for its array of object **id**s. It's initial capacity will be 0. Minimal amounts of memory will be allocated when objects are added to the List. Or an initial capacity can be set, before objects are added, using the **setAvailableCapacity:** method. Returns **self**.

See also: – **initCount:**, – **setAvailableCapacity:**

initCount:

– **initCount:**(unsigned int)*numSlots*

Initializes the receiver, a new List object, by allocating enough memory for it to hold *numSlots* objects. Returns **self**.

This method is the designated initializer for the class. It should be used immediately after memory for the List has been allocated and before any objects have been assigned to it; it shouldn't be used to reinitialize a List that's already in use.

See also: – **capacity**

insertObject:at:

– **insertObject:***anObject* **at:**(unsigned int)*index*

Inserts *anObject* into the List at *index*, moving objects down one slot to make room. If *index* equals the value returned by the **count** method, *anObject* is inserted at the end of the List. However, the insertion fails if *index* is greater than the value returned by **count** or *anObject* is **nil**.

If *anObject* is successfully inserted into the List, this method returns **self**. If not, it returns **nil**.

See also: – **count**, – **addObject:**

isEqual:

– (BOOL)**isEqual:***anObject*

Compares the receiving List to *anObject*. If *anObject* is a List with exactly the same contents as the receiver, this method returns YES. If not, it returns NO.

Two Lists have the same contents if they each hold the same number of objects and the **id**s in each List are identical and occur in the same order.

lastObject

– **lastObject**

Returns the last object in the List, or **nil** if there are no objects in the List. This method doesn't remove the object that's returned.

See also: – **removeLastObject**

makeObjectsPerform:

– **makeObjectsPerform:**(SEL)*aSelector*

Sends an *aSelector* message to each object in the List in reverse order (starting with the last object and continuing backwards through the List to the first object), and returns **self**. The *aSelector* method must be one that takes no arguments. It shouldn't have the side effect of modifying the List.

makeObjectsPerform:with:

– **makeObjectsPerform:**(SEL)*aSelector* **with:***anObject*

Sends an *aSelector* message to each object in the List in reverse order (starting with the last object and continuing backwards through the List to the first object), and returns **self**. The message is sent each time with *anObject* as an argument, so the *aSelector* method must be one that takes a single argument of type **id**. The *aSelector* method shouldn't, as a side effect, modify the List.

objectAt:

– **objectAt:**(unsigned int)*index*

Returns the **id** of the object located at slot *index*, or **nil** if *index* is beyond the end of the List.

See also: – **count**

read:

– **read:**(NXTypedStream *)*stream*

Reads the List and all the objects it contains from the typed stream *stream.*

See also: – **write:**

removeLastObject

– **removeLastObject**

Removes the object occupying the last position in the List and returns it. If there are no objects in the List, this method returns **nil**.

See also: – **lastObject**, – **removeObjectAt:**

removeObject:

– **removeObject:***anObject*

Removes the first occurrence of *anObject* from the List, and returns it. If *anObject* isn't in the List, this method returns **nil**.

The positions of the remaining objects in the List are adjusted so there's no gap.

See also: – **removeLastObject**, – **removeObjectAt:**

removeObjectAt:

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

Removes the object located at *index* and returns it. If there's no object at *index*, this method returns **nil**.

The positions of the remaining objects in the List are adjusted so there's no gap.

See also: – **removeLastObject**, – **removeObject:**

replaceObject:with:

– **replaceObject:***anObject* **with:***newObject*

Replaces the first occurrence of *anObject* in the List with *newObject*, and returns *anObject*. However, if *newObject* is **nil** or *anObject* isn't in the List, nothing is replaced and **nil** is returned.

See also: – **replaceObjectAt:with:**

replaceObjectAt:with:

– **replaceObjectAt:**(unsigned int)*index* **with:***newObject*

Returns the object at *index* after replacing it with *newObject*. If there's no object at *index* or *newObject* is **nil**, nothing is replaced and **nil** is returned.

See also: – **replaceObject:with:**

setAvailableCapacity:

– **setAvailableCapacity:**(unsigned int)*numSlots*

Sets the storage capacity of the List to at least *numSlots* objects and returns **self**. However, if the List already contains more than *numSlots* objects (if the **count** method returns a number greater than *numSlots*), its capacity is left unchanged and **nil** is returned.

See also: – **capacity**, – **count**

write:

– **write:**(NXTypedStream *)*stream*

Writes the List, including all the objects it contains, to the typed stream *stream*.

See also: – **read:**

NXBundle

Inherits From: Object

Declared In: objc/NXBundle.h

Class Description

An NXBundle is an object that corresponds to a directory where program resources are stored. The directory, in essence, "bundles" a set of resources used by an application, and the NXBundle object makes those resources available to the application. It's able to find requested resources in the directory and, in some cases, dynamically load executable code. The term "bundle" is used both for the object and for the directory it represents.

Bundled resources might include such things as:

- Images
- Sounds
- Character strings
- Nib files—files with a ".nib" extension—archived by Interface Builder
- Executable code

Each resource resides in a separate file.

Localized Resources

If an application is to be used in more than one part of the world, its resources may need to be customized—*localized*—for language, country, or cultural region. It may need, for example, to have separate Japanese, English, French, Hindi, and Swedish versions of the character strings that label menu commands. Its nib files might similarly need to be localized, as well as any images or sounds it uses.

The resource files specific to a particular language are grouped together in a subdirectory of the bundle directory. The subdirectory has the name of the language (in English) followed by a ".lproj" extension (for "language project"). The application mentioned above, for example, would have **Japanese.lproj**, **English.lproj**, **French.lproj**, **Hindi.lproj**, and **Swedish.lproj** subdirectories.

Each ".lproj" subdirectory in a bundle has the same set of files; all versions of a resource file must have the same name. Thus, **myIcon.tiff** in **French.lproj** should be the French counterpart to the Swedish **myIcon.tiff** in **Swedish.lproj**, and so on.

If two or more languages share the same localized version of a file, the file can be stored in just one of the ".lproj" subdirectories, while the other subdirectories keep (hard or soft) links to it. If a resource doesn't need to be localized at all, it's stored in the bundle directory itself, not in the ".lproj" subdirectores.

The user determines which set of localized resources will actually be used by the application. NXBundle objects rely on the language preferences set by the user in the Preferences application. Preferences lets users order a list of available languages so that the most preferred language is first, the second most preferred language is second, and so on.

When an NXBundle is asked for a resource file, it provides the path to the resource that best matches the user's language preferences. In the following code, for example, the application sends a **getPath:forResource:ofType:** message to ask for the path to the **myIcon.tiff** file. With the path in hand, it can use other facilities (here NXImage's **initFromFile:** method) to access the resource.

```
char        buf[MAXPATHLEN + 1];
NXBundle   *bundle;
NXImage    *image;

bundle = [NXBundle bundleForClass:[self class]];
if ( [bundle getPath:buf forResource:"myIcon" ofType:"tiff"] ) {
    image = [[NXImage alloc] initFromFile:buf];
    . . .
}
```

The Main Bundle

Every application is considered to have at least one bundle—its *main bundle*, the directory where its executable file is located. If the application is organized into a file package marked by a ".app" extension, the file package is the main bundle.

Note: A file package is a directory that the Workspace Manager presents to users as if it were a simple file; the contents of the directory are hidden. A file package for an application includes the application executable plus other files required by the application as it runs. It bears the same name as the executable file but adds a ".app" extension that identifies it to the Workspace Manager. For example, if you develop a Rutabaga application and place it in a **Rutabaga.app** directory with various ".nib" and TIFF files that the application will use, the **Rutabaga.app** directory is its file package and its main bundle.

Other Bundles

An application can be organized into any number of other bundles in addition to the main bundle. These other bundles usually reside inside the file package, but they can be located anywhere in the file system. Each bundle directory is represented in the application by a separate NXBundle object.

By convention, bundle directories other than the main bundle end in a ".bundle" extension, which instructs the Workspace Manager to hide the contents of the directory just as it hides the contents of a file package. The extension isn't required, but it's a good idea, especially if the bundle isn't already hidden by virtue of being inside a file package.

Dynamically Loadable Classes

Any bundle directory can contain a file with executable code. For the main bundle, that file is the application executable that's loaded into memory when the application is launched. The executable in the main bundle includes the **main**() function and other code necessary to start up the application.

Executable files in other bundle directories hold class (and category) definitions that the NXBundle object can dynamically load while the application runs. When asked, the NXBundle returns class objects for the classes (and categories) stored in the file. It waits to load the file until those classes are needed.

In the example below, the first line of code creates an instance of a class provided by an NXBundle object. If the class had not already been loaded into memory, asking for the class would cause it to be loaded.

```
id foo = [[[myBundle classNamed:"Reporter"] alloc] init];
if ( foo ) {
    [foo doSomething];
    . . .
}
```

By using a number of separate bundles in this way, you can split an application into smaller, more manageable pieces. Each piece is loaded into memory only when the code being executed requires it, so the application can start up faster than it otherwise would. And, assuming that only the rare user will exercise every part of the application, it will also consume less memory as it runs.

The file that contains dynamically loadable code must have the same name as the bundle directory, but without the ".bundle" extension.

Since each bundle can have only one executable file, that file should be kept free of localizable content. Anything that needs to be localized should be segregated into separate resource files and stored in ".lproj" subdirectories.

Instance Variables

None declared in this class.

Method Types

Initializing a new NXBundle object	– initForDirectory:
Getting and freeing an NXBundle	+ mainBundle + bundleForClass: – free
Getting a bundled class	– principalClass – classNamed:
Setting which resources to use	+ setSystemLanguages:
Finding a resource	– getPath:forResource:ofType: + getPath:forResource:ofType:inDirectory:withVersion:
Getting the bundle directory	– directory
Setting the version	– setVersion: – version

Class Methods

bundleForClass:

+ **bundleForClass:***classObject*

Returns the NXBundle object that dynamically loaded *classObject*, or the main bundle object if *classObject* was not dynamically loaded.

See also: + **mainBundle**

getPath:forResource:ofType:inDirectory:withVersion:

+ (BOOL)**getPath:**(char *)*path*
forResource:(const char *)*filename*
ofType:(const char *)*extension*
inDirectory:(const char *)*directory*
withVersion:(int)*version*

Returns YES if the specified resource file is available within the *directory*, and NO if it's not. If *path* is not NULL, a full pathname to the file is copied into the buffer it points to. To accommodate all possible pathnames, the *path* buffer should be at least MAXPATHLEN + 1 characters long. MAXPATHLEN is defined in the **sys/param.h** header file.

This method works just like the **getPath:forResource:ofType:** instance method, except that it searches for the resource in *directory* (rather than in the directory associated with the instance) and it tests against *version* (rather than the version last set by **setVersion:**). Therefore, if you only occasionally search for a resource in *directory* and don't need to dynamically load code from it, you can use this method instead of **getPath:forResource:ofType:** and avoid creating an NXBundle instance.

See also: – **getPath:forResource:ofType:**, – **setVersion:**, + **setSystemLanguages**:

mainBundle

+ **mainBundle**

Returns the NXBundle object that corresponds to the directory where the application executable (the file that's loaded into memory to start up the application) is located. This method allocates and initializes the NXBundle object, if it doesn't already exist.

In general, the main bundle corresponds to an application file package, a directory that bears the name of the application and is marked by a ".app" extension.

See also: + **bundleForClass:**

setSystemLanguages:

+ **setSystemLanguages:**(const char * const *)*languageArray*

Informs the NXBundle class of the user's language preferences, and returns **self**. The argument, *languageArray*, is a pointer to an ordered list of null-terminated character strings. Each string is the name of a language.

Language names used for ".lproj" subdirectories should match those set by this method. By convention, the names are in English. These are among the names currently in use:

English
French
German
Japanese
Spanish
Swedish

This method responds to a message sent by the Application Kit when the application first starts up; it's not necessary for your application to set the system languages.

Instance Methods

classNamed:

– **classNamed:**(const char *)*classname*

Returns the class object for the *classname* class, or **nil** if *classname* isn't one of the classes associated with the receiving NXBundle.

Before returning, this method ensures that any code in the bundle directory has been loaded into memory, so the *classname* class will be part of the executable image, if it's available to the NXBundle object.

See also: – **principalClass**

directory

– (const char *)**directory**

Returns a pointer to the full pathname of the receiver's bundle directory.

See also: – **initForDirectory:**

free

– **free**

Frees the receiving NXBundle, and returns **nil**. However, the main bundle can't be freed, and neither can any NXBundle with dynamically loaded code. If it can't free the object, this method returns **self**.

getPath:forResource:ofType:

– (BOOL)**getPath:**(char *)*path*
forResource:(const char *)*filename*
ofType:(const char *)*extension*

Returns YES if the specified resource file is available within the bundle, and NO if it's not. If *path* is not NULL, a full pathname to the file is copied into the buffer it points to. To accommodate all possible pathnames, the *path* buffer should be at least MAXPATHLEN + 1 characters long. MAXPATHLEN is defined in the **sys/param.h** header file.

To find the resource file, this method first looks inside the bundle directory for ".lproj" subdirectories that match the user's language preferences (as specified in the Preferences application). It searches for subdirectories in the order of user preference.

When it finds a ".lproj" subdirectory for a preferred language, the NXBundle first makes sure that the subdirectory version (as specified in a **version** file) matches the version last set by the **setVersion:** method. If the versions don't match or if the subdirectory doesn't contain the requested resource file, the NXBundle continues the search by looking for the ".lproj" subdirectory for the next most preferred language.

The search stops, and this method returns, as soon as the resource file is found. If the file can't be found in any ".lproj" subdirectory, the NXBundle looks for a nonlocalized version of it in the bundle directory.

If the *extension* doesn't repeat an extension already specified in the *filename*, it's added to *filename* before the search begins. The *extension* can be NULL, but *filename* can't be.

See also: + getPath:forResource:ofType:inDirectory:withVersion:, + setSystemLanguages:, – setVersion:

initForDirectory:

– **initForDirectory:**(const char *)*fullPath*

Initializes a newly allocated NXBundle object to make it the NXBundle for the *fullPath* directory. *fullPath* must be a full pathname for a directory.

If the directory doesn't exist or the user doesn't have access to it, the NXBundle is freed and this method returns **nil**. If the application already has an NXBundle object for the *fullPath* directory, this method frees the receiver and returns the existing object.

It's not necessary to allocate and initialize an object for the main bundle; the **mainBundle** method provides it.

See also: + mainBundle

principalClass

– **principalClass**

Returns the class object for a class that's dynamically loaded by the NXBundle, or **nil** if the NXBundle can't dynamically load any classes. Classes can be loaded from just one file within the bundle directory, a file that has the same name as the directory (but without the ".bundle" extension). If that file contains a single class, this method returns it. If the file contains more than one loadable class, this method returns the first one it encounters—that is, the first one listed on the **ld** command line that created the file. In the following example, Reporter would be the principal class:

```
ld -o myBundle -r Reporter.o NotePad.o QueryList.o
```

In general, the principal class should be the one that controls all the other classes that are dynamically loaded with it.

Before returning, this method ensures that any loadable code in the bundle directory has in fact been loaded into memory. If the NXBundle can load any classes at all, the principal class will be part of the executable image.

If the receiver is the main bundle object, this method returns **nil**. The main bundle doesn't have a principal class.

See also: – **classNamed:**

setVersion:

– **setVersion:**(int)*version*

Sets the version that the NXBundle will use when searching ".lproj" subdirectories for resource files, and returns **self**. The default version is 0.

See also: – **getPath:forResource:ofType:**, – **version**

version

– (int)**version**

Returns the version last set by the **setVersion:** method, or 0 if no version has been set.

See also: – **setVersion:**

NXStringTable

Inherits From: HashTable : Object

Declared In: objc/NXStringTable.h

Class Description

NXStringTable defines an object that associates a key with a value. Both the key and the value must be character strings. For example, these keys and values might be associated in a particular NXStringTable:

Key	Value
"Yes"	"Oui"
"No"	"Non"

By using an NXStringTable object to store your application's character strings, you can reduce the effort required to adapt the application to different language markets. Interface Builder give you direct access to NXStringTables, letting you create and initialize a string table and connect it into your application.

A new NXStringTable instance can be created either through Interface Builder's Classes window or through the inherited **alloc...** and **init...** methods. Similarly, you can establish the contents of an NXStringTable either directly through Interface Builder or programmatically through NXStringTable methods that read keys and values that are stored in a file (see **readFromFile:** and **readFromStream:**). Each assignment in the file can be of either of these formats:

```
"key" = "value";
"key";
```

If only *key* is present for a particular assignment, the corresponding value is taken to be identical to *key*.

A valid key or value—a valid token—is composed of text enclosed in double quotes. The text can't include double quotes (except in an escape sequence; see table) or the null character. It can include these escape sequences:

Escape Sequence	Meaning
\a	alert (bell)
\b	backspace
\f	formfeed
\n	newline
\r	carriage return
\t	horizontal tab
\v	vertical tab
\"	double quote

The backslash is stripped from any other character; consequently, numeric escape codes aren't interpreted. White space between tokens is ignored. A key or value can't exceed MAX_NXSTRINGTABLE_LENGTH characters.

The file can also include standard C-language comments; the NXStringTable ignores them. Comments can provide valuable information to a person who's translating or documenting the application.

To retrieve the value associated with a specific key, send a **valueForStringKey:** message to the NXStringTable. For example, assuming *myStringTable* is an NXStringTable containing the appropriate keys and values, this call would display an attention panel announcing a problem opening a file:

```
NXRunAlertPanel([myStringTable valueForStringKey:"openTitle"],
                [myStringTable valueForStringKey:"openError"],
                "OK",
                NULL,
                NULL);
```

If you're accessing NXStringTables through Interface Builder, please note the following. For efficiency, use several NXStringTables—each in its own interface file—rather than one large one. By using several NXStringTables, your application can load only those strings that it needs at a particular time. For example, you might place all the strings associated with a help system in an NXStringTable in one interface file and those associated with error messages in another NXStringTable in another file. When the user accesses the help system for the first time, the application can load the appropriate NXStringTable. Also, instantiate only one copy of any individual NXStringTable. Don't put an NXStringTable object in an interface file that will be loaded more than once, since multiple copies of the same table will result.

Instance Variables

None declared in this class.

Method Types

Initializing and freeing an NXStringTable	– init – free
Querying an NXStringTable	– valueForStringKey:
Reading and writing elements	– readFromFile: – writeToFile: – readFromStream: – writeToStream:

Instance Methods

free

– **free**

Frees the string table and its strings. You should never send a **freeObjects** (HashTable) message to an NXStringTable.

init

– **init**

Initializes a new NXStringTable. This is the designated initializer for the NXStringTable class. Returns **self**.

readFromFile:

– **readFromFile:**(const char *)*fileName*

Reads an ASCII representation of the NXStringTable's keys and values from *fileName*. The NXStringTable opens a stream on the file and then sends itself a **readFromStream:** message to load the data. See "Class Description" above for the format of the data. Returns **nil** on error; otherwise, returns **self**.

See also: – **readFromStream:**

readFromStream:

– **readFromStream:**(NXStream *)*stream*

Reads an ASCII representation of the NXStringTable's keys and values from *stream*. See "Class Description" above for the format of the data. Returns **nil** on error; otherwise, returns **self**.

See also: – **readFromFile:**

valueForStringKey:

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

Searches the string table for the value that corresponds to the key *aString*. Returns NULL if and only if no value is found for that key; otherwise, returns a pointer to the value.

writeToFile:

– **writeToFile:**(const char *)*fileName*

Writes an ASCII representation of the NXStringTable's keys and values to *fileName*. The NXStringTable opens a stream on the file and then sends itself a **writeToStream:** message. See "Class Description" above for the format of the data. Returns **nil** if an error occurs; otherwise, returns **self**.

See also: – **writeToStream:**

writeToStream:

– **writeToStream:**(NXStream *)*stream*

Writes an ASCII representation of the NXStringTable's keys and values to *stream*. See "Class Description" above for the format of the data. Returns **self**.

See also: – **writeToFile:**

Storage

Inherits From: Object

Declared In: objc/Storage.h

Class Description

The Storage class implements a general storage allocator. Each Storage object manages an array containing data elements of an arbitrary type. All the elements must be of the same type. When an element is added to the Storage object, it's copied into the array. The array grows dynamically when necessary; its capacity doesn't need to be explicitly adjusted.

Because a Storage object holds elements of an arbitrary type, you don't have to define a special class for each type of data you want to store. When setting up a new instance of the class, you specify the size of the elements and a description of their type. The type description is needed for archiving the object and must agree with the specified element size. It's encoded in a string using the descriptor codes listed in the table below:

Type	Code	Type	Code
int	i	**char**	c
unsigned int	I	**unsigned char**	C
short	s	**char** *	*
unsigned short	S	NXAtom	%
long	l	**id**	@
unsigned long	L	Class	#
float	f	SEL	:
double	d	structure	{<*types*>}
ignored	!	array	[<*count*><*types*>]

For example, "[15d]" means that each stored element is an array of fifteen **doubles**, and "{csi*@}" means that each stored element is a structure containing a **char**, a **short**, an **int**, a character pointer, and an object.

Most of these codes are identical to ones that would be returned by the **@encode()** compiler directive. However, there are some differences:

- A structure description can contain only encoded type information between the braces. It can't include a full type name or structure name.
- The '%' descriptor specifies a unique string pointer. When the pointer is unarchived, the **NXUniqueString()** function is called to make sure that it's also unique within the new context.
- The '!' descriptor marks data that won't be archived. Each occurrence of '!' instructs the archiver to skip data the size of an **int**.
- A few **@encode()** descriptors—such as the ones for pointers, bitfields, and undefined types—should not be used. Use only the codes shown in the table above.

Instance Variables

void ***dataPtr**;
const char ***description**;
unsigned int **numElements**;
unsigned int **maxElements**;
unsigned int **elementSize**;

dataPtr	A pointer to the data stored by the object.
description	A string encoding the type of data stored.
numElements	The number of elements actually in the Storage array.
maxElements	The total number of elements that can fit within currently allocated memory.
elementSize	The size of each element in the array.

Method Types

Initializing a new Storage instance
- init
- initCount:elementSize:description:

Copying and freeing Storage objects
- copyFromZone:
- free

Getting, adding, and removing elements
- addElement:
- insertElement:at:
- removeElementAt:
- removeLastElement
- replaceElementAt:with:
- empty
- elementAt:

Comparing Storage objects
- isEqual:

Managing the storage capacity and type
- count
- description
- setAvailableCapacity:
- setNumSlots:

Archiving
- read:
- write:

Instance Methods

addElement:

– **addElement:**(void *)*anElement*

Adds *anElement* at the end of the Storage array and returns **self**. The size of the array is increased if necessary.

See also: – **insertElement:at:**

copyFromZone:

– **copyFromZone:**(NXZone *)*zone*

Returns a new Storage object containing the same data as the receiver. The data and the object are both copied, and memory for both is taken from *zone*. However, the description string is not copied; the two objects share the same string.

See also: – **copy** (Object)

count

– (unsigned int)**count**

Returns the number of elements currently in the Storage array.

See also: – **setNumSlots:**

description

– (const char *)**description**

Returns the string encoding the data type of elements in the Storage array.

See also: – **initCount:elementSize:description:**

elementAt:

– (void *)**elementAt:**(unsigned int)*index*

Returns a pointer to the element at *index* in the Storage array. If no element is stored at *index* (*index* is beyond the end of the array), a NULL pointer is returned.

Before using the pointer that's returned, you must convert it into the appropriate type by a cast. The pointer can be used either to read the element at *index* or to alter it.

See also: – **replaceElementAt:with:**, – **insertElement:at:**

empty

– **empty**

Empties the Storage array of all its elements and returns **self**. The current capacity of the array remains unchanged; nothing is deallocated.

See also: – **free**

free

– **free**

Frees the Storage object and all the elements it contains. Pointers stored in the object will be freed, but the data they point to won't be (unless the data is also stored in the object). You might want to free the data before freeing the Storage object. The description string isn't freed.

See also: – **empty**

init

– **init**

Initializes the Storage object so that it's ready to store object **id**s. The initial capacity of the array isn't set. In general, it's better to store object **id**s in a List object. Returns **self**.

See also: – **initCount:elementSize:description:**, – **initCount:** (List)

initCount:elementSize:description:

– **initCount:**(unsigned int)*count*
elementSize:(unsigned int)*sizeInBytes*
description:(const char *)*string*

Initializes the Storage object so that it has *count* elements. Each element is of size *sizeInBytes* and of the type described by *string*. Memory for all the elements is set to 0. Returns **self**.

If *string* is NULL, the object won't be archivable. Once set, the description string should never be modified.

This method is the designated initializer for the class. It's used to initialize Storage objects immediately after they have been allocated; it should never be used to reinitialize a Storage object that's already been placed in use.

insertElement:at:

– **insertElement:**(void *)*anElement* **at:**(unsigned int)*index*

Puts *anElement* in the Storage array at *index*. All elements between *index* and the last element are shifted to make room. The size of the array is increased if necessary. Returns **self**.

See also: – **addElement:**, – **setNumSlots:**

isEqual:

– (BOOL)**isEqual:***anObject*

Compares the receiver with *anObject*, and returns YES if they're the same and NO if they're not. Two Storage objects are considered to be the same if they have the same number of elements and the elements at each position in the array match.

read:

– **read:**(NXTypedStream *)*stream*

Reads the Storage object and the data it stores from the typed stream *stream*. Where an archived string is represented by a '%' descriptor, the **NXUniqueString()** function is called to make sure that the string is unique within the new context.

See also: – **write:**

removeElementAt:

– **removeAt:**(unsigned int)*index*

Removes the element located at *index* from the Storage array and returns **self**. All elements between *index* and the last element are shifted to close the gap.

See also: – **removeLastElement**

removeLastElement

– **removeLastElement**

Removes the last element from the Storage array and returns **self**.

See also: – **removeElementAt:**

replaceElementAt:with:

– **replaceElementAt:**(unsigned int)*index* **with:**(void *)*anElement*

Replaces the data at *index* with the data pointed to by *anElement*. However, if no element is stored at *index* (*index* is beyond the end of the array), nothing is replaced. Returns **self**.

See also: – **elementAt:**, – **insertElement:at:**

setAvailableCapacity:

– **setAvailableCapacity:**(unsigned int)*numSlots*

Sets the storage capacity of the array to at least *numSlots* elements and returns **self**. If the array already contains more than *numSlots* elements, its capacity is left unchanged and **nil** is returned.

See also: – **setNumSlots:**, – **count**

setNumSlots:

– **setNumSlots:**(unsigned int)*numSlots*

Sets the number of elements in the Storage array to *numSlots* and returns **self**. If *numSlots* is greater than the current number of elements in the array (the value returned by **count**), the new slots will be filled with zeros. If *numSlots* is less than the current number of elements in the array, access to all elements with indices equal to or greater than *numSlots* will be lost.

If necessary, this method increases the capacity of the storage array so there's room for at least *numSlots* elements.

See also: – **setAvailableCapacity:**, – **count**

write:

– **write:**(NXTypedStream *)*stream*

Writes the Storage object and its data to the typed stream *stream*.

See also: – **read:**

Functions

NXAllocErrorData(), NXResetErrorData()

SUMMARY Manage the error data buffer

DECLARED IN objc/error.h

SYNOPSIS
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()** (Application Kit)

NXAtEOS() → See NXSeek()

NXChangeBuffer() → See NXStreamCreateFromZone()

NXClose()

SUMMARY Close a stream

DECLARED IN streams/streams.h

SYNOPSIS 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 opened in 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()

NXCompareHashTables() → See NXCreateHashTable()

NXCopyHashTable() → See NXCreateHashTable()

NXCopyStringBuffer() → See NXUniqueString()

NXCopyStringBufferFromZone() → See NXUniqueString()

NXCountHashTable() → See NXHashInsert()

NXCreateChildZone() → See NXCreateZone()

NXCreateHashTable(), NXCreateHashTableFromZone(), NXFreeHashTable(), NXEmptyHashTable(), NXResetHashTable(), NXCopyHashTable(), NXCompareHashTables(), NXPtrHash(), NXStrHash(), NXPtrIsEqual(), NXStrIsEqual(), NXNoEffectFree(), NXReallyFree()

SUMMARY Create and free a hash table

DECLARED IN objc/hashtable.h

SYNOPSIS

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()

NXCreateZone(), NXCreateChildZone(), NXMergeZone(), NXDefaultMallocZone(), NXZoneFromPtr(), NXDestroyZone()

SUMMARY Manage memory zones

DECLARED IN objc/zone.h

SYNOPSIS

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 **zone*)
NXZone *__NXDefaultMallocZone__(void)
NXZone *__NXZoneFromPtr__(void **ptr*)
void __NXDestroyZone__(NXZone **zone*)

DESCRIPTION These functions set up and manage the memory zones that are used to improve locality of reference. A zone is a region of memory from which functions like **NXZoneMalloc()** can allocate storage. A pointer to a zone is passed to the allocation function, which returns memory from the specified zone. Keeping related data structures together in the same zone reduces the amount of paging activity that otherwise would be required.

NXCreateZone() creates a new zone of *startSize* bytes that will grow and shrink by *granularity* bytes; it returns a pointer to the new zone. The zone will grow as needed as memory is allocated from it, and will shrink as memory is freed. Each time the function is called, it creates and returns a new zone.

Since the point of using zones is to keep data structures together on the same page, small multiples of **vm_page_size** (declared in **mach/mach_init.h**) are a good choice for both *startSize* and *granularity*. If these parameters are too large, the benefits of zone allocation can be defeated.

The parameter *canFree* determines whether memory, once allocated, can be freed within the zone. If *canFree* is NO, memory can't be freed and allocation from the zone will be as fast as possible; but you will need to destroy the zone to reclaim the memory.

NXCreateChildZone() creates a new zone that obtains memory from an existing zone, *parentZone*. 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. Typically, child zones are used to ensure that a group of data structures are packed together within a larger zone; successive allocations within the child zone are contiguous. The zone is created with a *startSize* sufficient for what it will contain; it can be smaller than a page size. After the allocations are complete, **NXMergeZone()** is called to merge the child zone back into its parent. All memory that was allocated and initialized within the child then resides within the parent zone.

NXDefaultMallocZone() returns the default zone, which is created automatically at startup. This is the zone used by the standard C **malloc()** function.

NXZoneFromPtr() returns the zone for the *ptr* block of memory, or NX_NOZONE if the block was not allocated from a zone. The pointer must be one that was returned by a prior call to an allocation function.

The macro **NXDestroyZone()** destroys a zone; all the memory from the zone is reclaimed.

RETURN **NXCreateZone()** and **NXCreateChildZone()** return a pointer to a new zone. **NXDefaultMallocZone()** returns a pointer to the default zone, and **NXZoneFromPtr()** returns the zone for the ptr block of memory. A return of NX_NOZONE indicates that the zone couldn't be created or doesn't exist.

SEE ALSO **NXZoneMalloc()**

NXDefaultExceptionRaiser(), NXSetExceptionRaiser(), NXGetExceptionRaiser()

SUMMARY Set and return an exception raiser

DECLARED IN objc/error.h

SYNOPSIS
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 **objc/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 NXCreateZone()

NXDefaultRead() → See NXStreamCreateFromZone()

NXDefaultWrite() → See NXStreamCreateFromZone()

NXDestroyZone() → See NXCreateZone()

NXEmptyHashTable() → See NXCreateHashTable()

NXEndOfTypedStream()

SUMMARY Determine whether there's more data to be read

DECLARED IN objc/typedstream.h

SYNOPSIS BOOL **NXEndOfTypedStream**(NXTypedStream **stream*)

DESCRIPTION This function 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 access to its members.)

RETURN **NXEndOfTypedStream()** returns TRUE if the read operation has reached the end of the stream and no more data is available to be read; returns 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()**

NXFilePathSearch()

SUMMARY Search for and read a file

DECLARED IN defaults/defaults.h

SYNOPSIS 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()

NXFlush()

SUMMARY Flush a stream

DECLARED IN streams/streams.h

SYNOPSIS 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

DECLARED IN objc/typedstream.h

SYNOPSIS 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()**

NXFreeHashTable() → See NXCreateHashTable()

NXFreeObjectBuffer() → See NXReadObjectFromBuffer()

NXGetc() → See NXPutc()

NXGetDefaultValue() → See NXRegisterDefaults()

NXGetMemoryBuffer() → See NXOpenMemory()

NXGetTempFilename()

SUMMARY Create a temporary file name

DECLARED IN defaults/defaults.h

SYNOPSIS 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 the unique file name it generates.

NXGetTypedStreamZone(), NXSetTypedStreamZone()

SUMMARY Set zones for streams

DECLARED IN objc/typedstream.h

SYNOPSIS 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 **NXCreateZone()** 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()

NXHashGet() → See NXHashInsert()

NXHashInsert(), NXHashInsertIfAbsent(), NXHashMember(), NXHashGet(), NXHashRemove(), NXCountHashTable(), NXInitHashState(), NXNextHashState()

SUMMARY Manipulate the elements of a hash table

DECLARED IN objc/hashtable.h

SYNOPSIS
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()

NXInitHashState() → See NXHashInsert()

NXIsAlNum() → See NXIsAlpha()

NXIsAlpha(), NXIsAlNum(), NXIsCntrl(), NXIsDigit(), NXIsGraph(), NXIsLower(), NXIsPrint(), NXIsPunct(), NXIsSpace(), NXIsUpper(), NXIsXDigit(), NXIsAscii()

SUMMARY Classify NeXTSTEP-encoded values

DECLARED IN appkit/NXCType.h

SYNOPSIS

int **NXIsAlpha**(unsigned int *c*)
int **NXIsAlNum**(unsigned int *c*)
int **NXIsUpper**(unsigned int *c*)
int **NXIsLower**(unsigned int *c*)
int **NXIsDigit**(unsigned int *c*)
int **NXIsXDigit**(unsigned int *c*)
int **NXIsSpace**(unsigned int *c*)
int **NXIsPunct**(unsigned int *c*)
int **NXIsPrint**(unsigned int *c*)
int **NXIsGraph**(unsigned int *c*)
int **NXIsCntrl**(unsigned int *c*)
int **NXIsAscii**(unsigned int *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 **ctype**(3) UNIX manual page), 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 whether *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()

NXIsSpace() → See NXIsAlpha()

NXIsUpper() → See NXIsAlpha()

NXIsXDigit() → See NXIsAlpha()

NXLoadLocalizedStringFromTableInBundle() → See NXLocalizedString()

NXLocalizedString(), NXLocalizedStringFromTable(), NXLocalizedStringFromTableInBundle(), NXLoadLocalizedStringFromTableInBundle()

SUMMARY Get localized versions of strings

DECLARED IN objc/NXBundle.h

SYNOPSIS

const char ***NXLocalizedString**(const char **key*, const char **value*, *comment*)
const char ***NXLocalizedStringFromTable**(const char **table*, const char **key*, const char **value*, *comment*)
const char ***NXLocalizedStringFromTableInBundle**(const char **table*, NXBundle **bundle*, const char **key*, const char **value*, *comment*)
const char ***NXLoadLocalizedStringFromTableInBundle**(const char **table*, NXBundle **bundle*, const char **key*, const char **value*)

DESCRIPTION

These three macros and one function select a localized string to display to the user. They each look up the *key* string in a table and return a matching string in a language of the user's preference. For example, if the key is "Cancel" and the user's preferred language is French, the string returned might be "Annuler"; if the user's preferred language is German, the same key might designate "Abbrechen". Users choose their preferred languages in the Preferences application.

To localize your application—to permit it to be run in more than one language—you must (1) keep the compiled code free of any user-visible strings and (2) provide resource files containing those strings in all the languages you're willing to support. Language-specific resources are kept in *Language*.**lproj** subdirectories of a *bundle* directory that can be managed by an NXBundle object. Most applications keep ".lproj" subdirectories in the file package that contains the application executable. This file package is a directory named after the application and assigned a ".app" extension. When it contains resource files and ".lproj" subdirectories, it's also known as the *main bundle*. An application can be organized into additional bundle directories, each with its own set of subdirectories, inside the main bundle. (See the description of the NXBundle class for more on bundle directories.)

Each ".lproj" subdirectory of a bundle bears the name of a language—such as **English.lproj**, **French.lproj**, or **German.lproj**—and stores resources specific to that language. Every resource file is repeated (with the identical name) in every subdirectory of the bundle. In addition to strings that are displayed to the user, localized resources include images, sounds, and nib files produced by Interface Builder.

One kind of resource in a ".lproj" subdirectory is a string table—identified by a ".strings" extension on the file name. Entries in a string table look like this,

```
[ /* comment */ ]
"key" [ = "value" ] ;
```

where the square brackets indicate that the comment and value are optional. The key is a string that's used to identify the entry; it must be unique within a file. The value is the localized string that's matched to the key. If the key and value strings are identical, the value string can be omitted. The comment is typically an explanation that would aid translators preparing correct versions of the string in other languages.

For example, an **English.lproj** subdirectory might contain a **my.strings** file with this entry:

```
/* unable to open a file; %s is the file name */
"open failure"="Can't open %s";
```

In **French.lproj**, the **my.strings** file might have this entry:

```
/* unable to open a file; %s is the file name */
"open failure"="Ouverture de %s impossible";
```

And in **German.lproj**, the entry could look like this:

```
/* unable to open a file; %s is the file name */
"open failure"="%s kann nicht geöffnet werden";
```

The **NXLoadLocalizedStringFromTableInBundle()** function searches for a localized version of the string designated by *key*. It looks only in the bundle directory managed by the *bundle* object and in the string table named *table*, which may or may not include the ".strings" extension. If *bundle* is nil, it looks in the main bundle; if *table* is NULL, it looks for a file named **Localizable.strings**.

The search starts with the ".lproj" subdirectory of the user's most preferred language and continues down the ordered list of language preferences until the *table* file is found. (If *table* occurs in every subdirectory, it should be found for the user's preferred language, provided the application is localized for that language.) If *table* can't be found in any ".lproj" subdirectory, the function looks for it in the bundle directory itself.

If a *key* entry is found in the string table, the function returns the matching value string (the string in the entry after the equal sign). If a value string is absent from the entry, it returns the key string. If the string table can't be found, or if the table lacks an entry for *key*, it returns the default *value* passed to the function as an argument. If *value* is NULL, it returns *key*.

The three macros are defined on the **NXLoadLocalizedStringFromTableInBundle()** function and do just what it does. However, they're preferred to the function since, in combination with the **genstrings** utility, they can aid in constructing string tables. **genstrings** searches for each occurrence of the macros in source code and constructs string table entries from the *key*, *value*, and *comment* arguments it finds. The *comment* argument can simply be information for translators who might render localized versions of the entry; it's discarded by the preprocessor and is not passed to the function. **genstrings** writes the entries into the *table* file, creating the file and adding the ".strings" extension if necessary. In the case of **NXLocalizedString()**, which doesn't have a *table* argument, it writes the results to the standard output For example, from this code,

```
char *s;
s = NXLocalizedStringFromTable("my", "open failure", "Can't open %s",
unable to open a file; %s is the file name);
```

genstrings would construct the string table entry illustrated earlier and put it in the **my.strings** file. The **genstrings** utility is more fully documented on-line, in the file **Localization.rtfd** under the **/NextLibrary/Documentation/NextDev/Concepts** directory.

The **NXLocalizedStringFromTableInBundle()** macro works just like the **NXLoadLocalizedStringFromTableInBundle()** function, except that it provides source material for **genstrings**. The **NXLocalizedStringFromTable()** macro looks for the *key* string in the *table* file in the main bundle. The **NXLocalizedString()** macro, the simplest of the three to use, looks for the *key* string in the string table named **Localizable.string** in the main bundle.

RETURN The function and all three macros return a localized string designated by *key*, or *value* if the string can't be found, or *key* if the string can't be found and *value* is NULL.

NXLocalizedStringFromTable() → See NXLocalizedString()

NXLocalizedStringFromTableInBundle() → See NXLocalizedString()

NXMallocCheck(), NXNameZone(), NXZonePtrInfo()

SUMMARY Aid in debugging memory allocation

DECLARED IN objc/zone.h

SYNOPSIS
int **NXMallocCheck**(void)
void **NXNameZone**(NXZone *_zone_, const char *_name_)
void **NXZonePtrInfo**(void *_ptr_)

DESCRIPTION These functions assist in debugging memory allocation problems. **NXMallocCheck()** verifies all internal memory-allocation information, and returns 0 if there are no inconsistencies or errors. This function is used by **malloc_debug()**. **NXNameZone()** assigns *name* to *zone*. **NXZonePtrInfo()** prints various information about the *ptr* memory block to the standard output. The information includes the name of the zone, if one was assigned by **NXNameZone()**.

SEE ALSO **NXZoneMalloc()**, **NXCreateZone()**

NXMapFile() → See NXOpenMemory ()

NXMergeZone() → See NXCreateZone()

NXNameZone() → See NXMallocCheck()

NXNextHashState() → See NXHashInsert()

NXNoEffectFree() → See NXCreateHashTable()

NXOpenFile(), NXOpenPort()

SUMMARY Open a file stream or a Mach port stream

DECLARED IN streams/streams.h

SYNOPSIS 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 *NeXTSTEP Operating System Software* manual 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

DECLARED IN streams/streams.h

SYNOPSIS
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. If you try to map a file that doesn't exist, this function returns a NULL stream.

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()**. 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. **NXCloseMemory()** doesn't free the internal buffer: Use **NXGetMemoryBuffer()** to get the internal buffer and use **vm_deallocate()** to free it.

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

DECLARED IN objc/typedstream.h

SYNOPSIS
NXTypedStream *__NXOpenTypedStream__(NXStream *_stream_, int _mode_)
void __NXCloseTypedStream__(NXTypedStream *_stream_)
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.

Note: The *filename* argument to **NXOpenTypedStreamForFile()** is stored as a pointer. If the file is opened in NX_WRITEONLY mode, the referenced file isn't actually opened for writing until **NXCloseTypedStream()** is called. Thus if the string pointed to by *filename* changes between these two function calls, the data will be written to the file of the new name. **NXCloseTypedStream()** will raise an exception if *filename* can't be opened for writing.

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.

SEE ALSO **NXOpenMemory()**, **NXOpenFile()**, **NXClose()**, **NXCloseMemory()**, **NXReadType()**, **NXReadObject()**, **NXReadPoint()**

NXOpenTypedStreamForFile() → See NXOpenTypedStream()

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

DECLARED IN streams/streams.h

SYNOPSIS
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

DECLARED IN streams/streams.h

SYNOPSIS int **NXRead**(NXStream **stream*, void **buf*, int *count*)
int **NXWrite**(NXStream **stream*, const void **buf*, int *count*)

DESCRIPTION These macros 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.)

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 macros 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

DECLARED IN objc/typedstream.h

SYNOPSIS void **NXReadArray**(NXTypedStream **stream*, const char **dataType*, int *count*, void **data*)
void **NXWriteArray**(NXTypedStream **stream*, const char **dataType*, int *count*, const 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 in the "See Also" section.

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 access to its members.)

NXReadArray() and **NXWriteArray()** read and write an array of *count* elements of type *dataType* from or to *stream*. **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 the mode 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()**

NXReadDefault() → See NXRegisterDefaults()

NXReadObject(), NXWriteObject(), NXWriteObjectReference(), NXWriteRootObject()

SUMMARY Read or write Objective C objects from or to a typed stream

DECLARED IN objc/typedstream.h

SYNOPSIS
id **NXReadObject**(NXTypedStream *_stream_)
void **NXWriteObject**(NXTypedStream *_stream_, id _object_)
void **NXWriteObjectReference**(NXTypedStream *_stream_, id _object_)
void **NXWriteRootObject**(NXTypedStream *_stream_, 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 *stream*. 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 *stream* 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()** (Application Kit), and **NXTypedStreamClassVersion()**

NXReadObjectFromBuffer(), NXReadObjectFromBufferWithZone(), NXWriteRootObjectToBuffer(), NXFreeObjectBuffer()

SUMMARY Read and write an object to a typed-stream memory buffer

DECLARED IN objc/typedstream.h

SYNOPSIS
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.
NXReadObjectFromBufferWithZone() 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() → See NXReadObjectFromBuffer()

NXReadType(), NXWriteType(), NXReadTypes(), NXWriteTypes()

SUMMARY Read or write arbitrary data to a typed stream

DECLARED IN objc/typedstream.h

SYNOPSIS
void **NXReadType**(NXTypedStream **stream*, const char **type*, void **data*)
void **NXWriteType**(NXTypedStream **stream*, const char **type*, const void **data*)
void **NXReadTypes**(NXTypedStream **stream*, const char **types*, ...)
void **NXWriteTypes**(NXTypedStream **stream*, 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 in the "See Also" section.

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 data types and format string characters listed below are supported.

Type	Code	Type	Code
int	i	**char**	c
unsigned int	I	**unsigned char**	C
short	s	**char** *	*
unsigned short	S	NXAtom	%
long	l	**id**	@
unsigned long	L	Class	#
float	f	SEL	:
double	d	structure	{<*types*>}
ignored	!	array	[<*count*><*types*>]

For example, "[15d]" means that each stored element is an array of fifteen **doubles**, and "{csi*@}" means that each stored element is a structure containing a **char**, a **short**, an **int**, a character pointer, and an object.

Most of these codes are identical to ones that would be returned by the **@encode()** compiler directive. However, there are some differences:

- A structure description can contain only encoded type information between the braces. It can't include a full type name or structure name.
- The '%' descriptor specifies a unique string pointer. When the pointer is unarchived, the **NXUniqueString()** function is called to make sure that it's also unique within the new context.
- The '!' descriptor marks data that won't be archived. Each occurrence of '!' instructs the archiver to skip data the size of an **int**.
- A few **@encode()** descriptors—such as the ones for pointers, bitfields, and undefined types—should not be used. Use only the codes shown in the table above.

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);
```

Note: NXWriteType()/NXReadType() and **NXWriteTypes()/NXReadTypes()** must be used symmetrically. That is, if you write data values using a series of **NXWriteType()** function calls, you must read those values using a corresponsing series of **NXReadType()** function calls. A similar stricture applies for **NXWriteTypes()** and **NXReadTypes()**.

Note: Use **NXWriteType()** and **NXReadType()** to archive structures; for example, use the following code to write a structure of four floats:

```
NXWriteType(s, "{4f}", &data)
```

then use the corresponding code to read the structure:

```
NXReadType(s, "{4f}", &data)
```

Use the **NXWriteArray()** and **NXReadArray()** functions to write and read arrays. Avoid using the **NXWriteTypes()** and **NXReadTypes()** functions for structures and arrays; these functions can archive arrays and structures incorrectly and cause errors at runtime.

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()

NXReallyFree() → See NXCreateHashTable()

NXRegisterDefaults(), NXGetDefaultValue(), NXReadDefault(), NXSetDefault(), NXWriteDefault(), NXWriteDefaults(), NXUpdateDefault(), NXUpdateDefaults(), NXRemoveDefault(), NXSetDefaultsUser()

SUMMARY Set or read default values

DECLARED IN defaults/defaults.h

SYNOPSIS
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 **NXSetDefault**(const char *_owner_, const char *_name_, const char *_value_)
int **NXWriteDefault**(const char *_owner_, const char *_name_, const char *_value_)
int **NXWriteDefaults**(const char *_owner_, NXDefaultsVector _vector_)
const char ***NXUpdateDefault**(const char *_owner_, const char *_name_)
void **NXUpdateDefaults**(void)
int **NXRemoveDefault**(const char *_owner_, const char *_name_)
const char ***NXSetDefaultsUser**(const char *_newUser_)

DESCRIPTION These functions give you access to a system of *default parameters* through which you can allow users to customize your application to meet their needs. For example, you can allow users to determine what units of measurement your application will display or how often documents will be automatically saved. The parameters get the name *default* since they're commonly used to determine an application's default state at startup or the way it will act by default.

Parameter values can be set from the command line or from a user's *defaults database*. Since values are stored specific to a particular user, you can use the parameters to record user preferences or to capture the application's state in one session so that it can be carried over to the next session. You can invent whatever parameters your application needs. Some parameters are defined in NeXTSTEP software—for example, many record choices the user makes in the Preferences application. See Appendix B, "Default Parameters," for a listing of system-defined parameters that you can read or set.

Parameters are set on the command line by preceding the parameter name with a hyphen and following it with a value. For example, the following instruction would launch the Edit application on the host machine named "earth" and assign that name as the value of the NXHost parameter:

```
localhost> /NextApps/Edit.app/Edit -NXHost earth
```

Listing a parameter on the command line doesn't put it in the defaults database. To put a parameter in the defaults database, you must use the functions described below.

A defaults database is created automatically for each user. It's named **.NeXTdefaults** and is located in the **.NeXT** directory in the user's home directory. Each parameter in the database is made up of three components:

- An owner, which is either the name of a specific application or "GLOBAL"
- The name of the parameter
- The value associated with the parameter

Each component is specified as a character string.

At run time, the parameters your application will use are placed in an internal cache. By using this cache, you avoid having to open the user's defaults database each time that you need access to a parameter. The cache is a list of parameters containing the same three components for each parameter as the database: the owner, the parameter name, and the associated value.

To register parameters in the cache, call **NXRegisterDefaults()** and give it two arguments: a character string specifying the owner and an NXDefaultsVector array. This array is a list of structures, each containing a parameter name and a value. (NXDefaultsVector is defined in the header file **defaults/defaults.h**.) Every application should register default parameters early in the program, before any of the values are needed.

Note: You should use the full market name of your product as the owner of the parameters you create. This will avoid conflicts with existing parameters. Noncommercial applications might use the name of the program and the author or institution.

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 **ArbDefaults** for the owner "Arboretum" (note that NULL is used to signal the end of the NXDefaultsVector array):

```
+ initialize
{
    static NXDefaultsVector ArbDefaults = {
        {"NXMeasurementUnit", NULL},
        {"AutoPropagate", "YES"},
        {NULL}
    };

    NXRegisterDefaults("Arboretum", ArbDefaults);

    return self;
}
```

If the defaults database doesn't exist when **NXRegisterDefaults()** is called, it's automatically created and placed in the **.NeXT** directory; the directory is created if necessary.

NXRegisterDefaults() creates a cache that contains a value for each of the parameters listed in the NXDefaultsVector array. For each parameter, a value is determined by first looking to see if it was defined on the command line (if the application was launched that way); if not, the user's defaults database (**.NeXTdefaults**) is searched. If **NXRegisterDefaults()** finds a parameter and owner in the database that match those passed to it as arguments, the corresponding value from the database is placed in the cache. If no parameter-owner match is found, **NXRegisterDefaults()** searches the database's global parameters—those owned by "GLOBAL"—for a matching parameter, and, if it finds one, places the corresponding value in the cache. If a match still isn't found, the parameter-value pair listed in the NXDefaultsVector array is used. (A value can be specified in the array as NULL.)

To summarize, this is the precedence ordering used to obtain a value for a given parameter for the cache:

1. The command line
2. The user's defaults database (**.NeXTdefaults**), with a matching owner
3. The user's defaults database, with the owner listed as "GLOBAL"
4. The NXDefaultsVector array passed to **NXRegisterDefaults()**

To read a parameter value, you'll most often call **NXGetDefaultValue()**. This function takes an owner and name of a parameter as arguments and returns a character pointer to the value for that parameter. **NXGetDefaultValue()** first looks in the cache for a matching owner-parameter item. If **NXGetDefaultValue()** doesn't find a match in the cache (which would be the case only if the parameter wasn't in the NXDefaultsVector array passed to **NXRegisterDefaults()**), it searches the user's defaults database (**.NeXTdefaults**) for the owner and parameter. If still no match is found, it searches for a matching global parameter, first in the cache and then in the database. If the value is found in the database rather than the cache, **NXGetDefaultValue()** registers that value for subsequent use.

Occasionally, you may want to search only the database for a parameter value and ignore the cache. For example, you might want to get a parameter value that another application may have changed after the cache was created. In these rare cases, call **NXReadDefault()**, which takes an owner and parameter name 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 character pointer to the 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 cache with **NXSetDefault()**.

NXSetDefault() takes as arguments an owner, the name of a parameter, and a value for that parameter. The parameter and its value are placed in the cache, but they aren't written into the user's defaults database (**.NeXTdefaults**).

NXWriteDefault() writes the owner, parameter, and value specified as its arguments into the user's defaults database and places them in the cache. Similarly, **NXWriteDefaults()** writes a vector of parameters 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 internal cache might not agree on the value of a given parameter. You can update the cache with any changes that have been made to the database since the cache was created by calling **NXUpdateDefault()** or **NXUpdateDefaults()**. Both functions compare the cache and the database. If a value is found in the database that is newer than the corresponding value in the internal cache, the new value is written into the cache.

NXUpdateDefault() updates the value for the single parameter and owner given as its arguments. **NXUpdateDefaults()**, which takes no arguments, updates the entire cache. It checks every parameter in the cache, determines whether a newer value exists in the database, and puts any newer values it finds in the cache.

NXRemoveDefault() removes the specified owner-parameter pair from both the user database and the internal cache.

Ordinarily, the functions described above use the database belonging to the user who started the application. **NXSetDefaultsUser()** changes which defaults database is used by subsequent calls to these functions. **NXSetDefaultsUser()** accepts the name of a user whose database you wish to use; 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 internal cache are purged; use **NXGetDefaultValue()** or **NXRegisterDefaults()** to get the new user's defaults for your application. When **NXSetDefaultsUser()** 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 character pointer to the requested parameter value, or 0 if the database couldn't be opened.

NXReadDefault() returns a character pointer to the parameter value; if a value is not found, NULL is returned.

NXSetDefault() returns 1 if it successfully set a parameter value, and 0 if not.

NXWriteDefault() returns 1 unless an error occurs while writing the parameter value, in which case it returns 0.

NXWriteDefaults() returns the number of successfully written parameter values.

NXUpdateDefault() returns the new value, or NULL if the value did not need to be updated.

NXRemoveDefault() returns 1, or 0 if the parameter couldn't be removed.

NXSetDefaultsUser() returns the login name of the user whose defaults database was being used before the function was called.

NXRegisterPrintfProc()

SUMMARY Register a procedure for formatting data written to a stream

DECLARED IN streams/streams.h

SYNOPSIS 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 *procData)
```

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 **tabOver()**. **NXPrintf()** then passes the variable **itemOne** to **tabOver()** for formatting, which formats the item and places it in **myStream**.

SEE ALSO **NXPutc()**

NXRemoveDefault() → See NXRegisterDefaults()

NXResetErrorData() → See NXAllocErrorData()

NXResetHashTable() → See NXCreateHashTable()

NXSaveToFile() → See NXOpenMemory()

NXScanf() → See NXPutc()

NXSeek(), NXTell(), NXAtEOS()

SUMMARY Set or report current position in a stream

DECLARED IN streams/streams.h

SYNOPSIS
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.

NXSetDefault() → See NXRegisterDefaults()

NXSetDefaultsUser() → See NXRegisterDefaults()

NXSetExceptionRaiser() → See NXDefaultExceptionRaiser()

NXSetTypedStreamZone() → See NXGetTypedStreamZone()

NXSetUncaughtExceptionHandler(), NXGetUncaughtExceptionHandler()

SUMMARY Handle uncaught exceptions

DECLARED IN objc/error.h

SYNOPSIS 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 **objc/error.h**.

SEE ALSO **NX_RAISE()**, **NXDefaultTopLevelErrorHandler()**

NXStreamCreateFromZone(), NXStreamCreate(), NXStreamDestroy(), NXDefaultRead(), NXDefaultWrite(), NXFill(), NXChangeBuffer()

SUMMARY Support a user-defined stream

DECLARED IN streams/streamsimpl.h

SYNOPSIS

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 *NeXTSTEP Programming Interface 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 **NXCreateZone()** 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 NXStreamCreateFromZone()

NXStrHash() → See NXCreateHashTable()

NXStrIsEqual() → See NXCreateHashTable()

NXTell() → See NXSeek()

NXToAscii(), NXToLower(), NXToUpper()

SUMMARY Convert NeXTSTEP-encoded characters

DECLARED IN NXCType.h

SYNOPSIS
unsigned char ***NXToAscii**(unsigned int *c*)
int **NXToLower**(unsigned int *c*)
int **NXToUpper**(unsigned int *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 **ctype**(3) UNIX manual page), which operate on characters in the ASCII character set.

NXToLower() converts an uppercase letter to its lowercase equivalent, and **NXToUpper()** converts a lowercase letter to its uppercase 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 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	.

Extended Character	Converts to
brokenbar	|
bullet	*
ellipsis	...
questiondown	?
onesuperior	1
twosuperior	2
threesuperior	3
emdash	-
plusminus	+-
onequarter	1/4
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()

NXToUpper() → See NXToAscii()

NXTypedStreamClassVersion()

SUMMARY Get the class version number of an archived instance

DECLARED IN objc/typedstream.h

SYNOPSIS int **NXTypedStreamClassVersion**(NXTypedStream **stream*, 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:***stream*] message and before performing version-specific initialization. Calling this function doesn't change the position of the read pointer in *stream*. 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 nonzero 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
{
    if (self == [MyClass class]) {
      [MyClass setVersion:MYCLASS_CURRENT_VERSION];
    }
    return self;
}
```

Note that this code tests to see that initialize is being invoked by the implementing class, not a subclass. This is useful to assure that subclasses don't inherit the version number (or other class-specific details).

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 access to its members.

SEE ALSO **NXReadObject()**

NXUngetc() → See NXPutc()

NXUniqueString(), NXUniqueStringWithLength(), NXUniqueStringNoCopy(), NXCopyStringBuffer(), NXCopyStringBufferFromZone()

SUMMARY Manipulate a string buffer

DECLARED IN objc/hashtable.h

SYNOPSIS

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 functions allocate memory for and return 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 the **objc/hashtable.h** header file.)

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()

NXUpdateDefault() → See NXRegisterDefaults()

NXUpdateDefaults() → See NXRegisterDefaults()

NXVPrintf() → See NXPutc()

NXVScanf() → See NXPutc()

NXWrite() → See NXRead()

NXWriteArray() → See NXReadArray()

NXWriteDefault() → See NXRegisterDefaults()

NXWriteDefaults() → See NXRegisterDefaults()

NXWriteObject() → See NXReadObject()

NXWriteObjectReference() → See NXReadObject()

NXWriteRootObject() → See NXReadObject()

NXWriteRootObjectToBuffer() → See NXReadObjectFromBuffer()

NXWriteType() → See NXReadType()

NXWriteTypes() → See NXReadType()

NXZoneCalloc() → See NXZoneMalloc()

NXZoneFromPtr() → See NXCreateZone()

NXZoneFree() → See NXZoneMalloc()

NXZoneMalloc(), NXZoneCalloc(), NXZoneRealloc(), NXZoneFree()

SUMMARY Allocate and free memory within a zone

DECLARED IN objc/zone.h

SYNOPSIS

void ***NXZoneMalloc**(NXZone **zone*, size_t *size*)
void ***NXZoneCalloc**(NXZone **zone*, size_t *numElems*, size_t *numBytes*)
void ***NXZoneRealloc**(NXZone **zone*, void **ptr*, size_t *size*)
void **NXZoneFree**(NXZone **zone*, void **ptr*)

DESCRIPTION These functions allocate and free memory within a particular region, or *zone*. They're similar to the standard C library functions **malloc()**, **calloc()**, **realloc()**, and **free()**, but allow more control over memory placement. By placing data structures that are likely to be used in conjunction with each other in the same zone, you can ensure better locality of reference. This can significantly improve performance on a paged virtual memory system. When related data structures are grouped close together, consecutive references are less likely to result in memory paging activity.

To use these functions, you must first obtain a pointer to a zone, generally by creating a new zone using **NXCreateZone()**. The zone pointer is passed as the first argument to each of these functions. Memory is allocated from the zone specified.

NXZoneMalloc() allocates *size* bytes from *zone*, and returns a pointer to the allocated memory. **NXZoneCalloc()** allocates enough memory from *zone* for *numElems* elements, each with a size of *numBytes* bytes, and returns a pointer to the allocated memory. Both allocate memory that's aligned to accommodate any C data type. Like **calloc()**, **NXZoneCalloc()** sets the allocated memory to 0 throughout; **NXZoneMalloc()**, like **malloc()**, does not.

NXZoneRealloc() changes the size of the block of memory pointed to by *ptr* to *size* bytes. It may allocate new memory to replace the old. If so, it moves the contents of the old memory block to the new block, up to a maximum of *size* bytes.

NXZoneFree() returns memory to the zone from which it was allocated. The standard C function **free()** does the same, but spends time finding which zone the memory belongs to.

For both **NXZoneRealloc()** and **NXZoneFree()**, *ptr* must be a pointer to a memory block that was returned by **NXZoneMalloc()**, **NXZoneCalloc()**, **NXZoneRealloc()**, or their standard C counterparts. The *zone* must be the one from which the *ptr* memory block was allocated; if it's not, the results are unpredictable, and possibly disastrous.

NXZoneMalloc(), **NXZoneRealloc()**, and **NXZoneFree()** are implemented as macros.

RETURN If successful, **NXZoneMalloc()**, **NXZoneCalloc()**, and **NXZoneRealloc()** return a pointer to the memory allocated (or reallocated). If unsuccessful, they return NULL.

SEE ALSO **NXCreateZone()**, **– allocFromZone:** (Object class)

NXZonePtrInfo() → See NXMallocCheck()

NXZoneRealloc() → See NXZoneMalloc()

NX_ADDRESS()

SUMMARY Get a pointer to the objects stored in a List

DECLARED IN objc/List.h

SYNOPSIS 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 List class

NX_ENDHANDLER → See NX_DURING

NX_DURING, NX_HANDLER, NX_ENDHANDLER

SUMMARY Mark exception handling domains and handlers

DECLARED IN objc/error.h

SYNOPSIS
NX_DURING
NX_HANDLER
NX_ENDHANDLER

DESCRIPTION These macros are used to delimit portions of code that are under the control of the NeXTSTEP exception handling system. Code that lies between the NX_DURING and NX_HANDLER macros is said to lie in an exception-handling domain. Code that lies between NX_HANDLER and NX_ENDHANDLER is said to be within the exception handler. A call to **NX_RAISE()** within the exception-handling domain transfers program execution to the first line of code in the exception handler. See **ExceptHandling.rtfd** in **/NextLibrary/Documentation/NextDev/Concepts** for more information.

SEE ALSO **NX_RAISE()**

NX_HANDLER → See NX_DURING

NX_RAISE(), NX_RERAISE(), NX_VALRETURN(), NX_VOIDRETURN

SUMMARY Raise an exception

DECLARED IN objc/error.h

SYNOPSIS 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 **objc/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()** (Application Kit), **NXRegisterErrorReporter()** (Application Kit), **NXDefaultExceptionRaiser()**

NX_RERAISE() → See NX_RAISE()

NX_VALRETURN() → See NX_RAISE()

NX_VOIDRETURN → See NX_RAISE()

Types and Constants

Defined Types

NXAtom

DECLARED IN objc/hashtable.h

SYNOPSIS

typedef const char ***NXAtom**;

DESCRIPTION NXAtom is the type for a unique string. A unique string is a string that is allocated once and for all (that is, never deallocated) and that has only one representation. Unique strings can therefore be compared using the equality operator (==) rather than using **strcmp()**. A unique string should never be modified (and in fact some memory protection is done to ensure that it won't be modified). To more explicitly declare that the string has been made unique, this synonym of **const char *** has been added.

SEE ALSO **NXUniqueString()**

NXDefaultsVector

DECLARED IN defaults/defaults.h

SYNOPSIS

```
typedef struct _NXDefault {
        char *name;
        char *value;
} NXDefaultsVector[];
```

DESCRIPTION This structure is used by the functions **NXRegisterDefaults()** and **NXWriteDefaults()**. It provides a way to specify an open-ended list of default name/value pairs as an argument to these functions.

NXExceptionRaiser

DECLARED IN objc/error.h

SYNOPSIS
```
typedef void NXExceptionRaiser(int code,
    const void *data1,
    const void *data2);
```

DESCRIPTION This type is used for the function that handles exceptions raised within an exception-handling domain. In NeXTSTEP, this function is by default **NXDefaultExceptionRaiser()**.

SEE ALSO **NXDefaultExceptionRaiser()**

NXHandler

DECLARED IN objc/error.h

SYNOPSIS
```
typedef struct _NXHandler {
    jmp_buf jumpState;
    struct _NXHandler *next;
    int code;
    const void *data1, *data2;
} NXHandler;
```

DESCRIPTION This structure is used by the NeXTSTEP exception-handling system to mark nodes in the chain of exception handlers. Its fields are:

jumpState	Place to jump to using **longjmp()**
next	Pointer to next exception handler
code	Error code of exception
data1	User-defined data about the exception
data2	User-defined data about the exception

SEE ALSO **NX_RAISE()**

NXHashState

DECLARED IN objc/hashtable.h

SYNOPSIS

```
typedef struct {
    int i;
    int j;
} NXHashState;
```

DESCRIPTION This type is used for the marker passed between the functions **NXInitHashState()** and **NXNextHashState()**. Its fields may change in the future, so your code shouldn't rely on the composition of an NXHashState structure.

SEE ALSO **NXInitHashState()** and **NXNextHashState()**

NXHashTable

DECLARED IN objc/hashtable.h

SYNOPSIS

```
typedef struct {
    const NXHashTablePrototype *prototype;
    unsigned count;
    unsigned nbBuckets;
    void *buckets;
    const void *info;
} NXHashTable;
```

DESCRIPTION This type is used to identify a hash table, such as the ones returned by **NXCreateHashTable()**. Its fields are private and shouldn't be accessed.

SEE ALSO **NXCreateHashTable()**

NXHashTablePrototype

DECLARED IN objc/hashtable.h

SYNOPSIS

```
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;
```

DESCRIPTION This type is used as one of the arguments to **NXCreateHashTable()**. Its fields specify the functions to be used for hashing, comparing, and freeing data elements:

hash	Identifies the hashing function
isEqual	Identifies the comparison function
free	Identifies the function that frees a data element
style	Reserved for future use

SEE ALSO **NXCreateHashTable()**

NXUncaughtExceptionHandler

DECLARED IN objc/error.h

SYNOPSIS

```
typedef void NXUncaughtExceptionHandler(int code,
    const void *data1,
    const void *data2);
```

DESCRIPTION This type is used for the function that handles exceptions raised outside of an exception-handling domain. In NeXTSTEP, this function can be set using **NXSetUncaughtExceptionHandler()**.

SEE ALSO **NXSetUncaughtExceptionHandler()**

NXZone

DECLARED IN objc/zone.h

SYNOPSIS

```
typedef struct _NXZone {
        void *(*realloc)(struct _NXZone *zonep, void *ptr, size_t size);
        void *(*malloc)(struct _NXZone *zonep, size_t size);
        void (*free)(struct _NXZone *zonep, void *ptr);
        void (*destroy)(struct _NXZone *zonep);
} NXZone;
```

DESCRIPTION This structure is used to identify and manage memory zones. The fields of the structure are private and subject to change in future releases; they should not be directly accessed or altered. Use **NXCreateZone()** or a similar function to establish a new zone.

SEE ALSO **NXCreateZone()** and **NXZoneMalloc()**

Symbolic Constants

List Constants

DECLARED IN objc/List.h

SYNOPSIS NX_NOT_IN_LIST

DESCRIPTION This constant is returned by List's **indexOf:** method when it can't find the object it's passed anywhere in the List.

NXStringTable Constants

DECLARED IN objc/NXStringTable.h

SYNOPSIS MAX_NXSTRINGTABLE_LENGTH 1024

DESCRIPTION This constant defines the maximum length for keys or values within an NXStringTable object.

Zone Constants

DECLARED IN objc/zone.h

SYNOPSIS NX_NOZONE (NXZone *)0

DESCRIPTION This constant is used as a return value by **NXCreateChildZone()**, **NXZoneFromPtr()**, and other functions to indicate the absence of a zone.

Global Variables

Command Line Arguments

DECLARED IN defaults/defaults.h

SYNOPSIS
extern int **NXArgc**;
extern char **NXArgv;

DESCRIPTION These global variables pass command-line arguments to a program when it begins executing. **NXArgc** is the number of command-line arguments the program was invoked with. **NXArgv** is a pointer to an array of character strings that contain the arguments, one per string.

HashTable Prototypes

DECLARED IN objc/hashtable.h

SYNOPSIS
const NXHashTablePrototype **NXPtrPrototype**;
const NXHashTablePrototype **NXStrPrototype**;
const NXHashTablePrototype **NXPtrStructKeyPrototype**;
const NXHashTablePrototype **NXStrStructKeyPrototype**;

DESCRIPTION These global variables identify hash table prototypes suitable for use with **NXCreateHashTable()**. The first two are used for hash tables of pointers and strings, respectively. They use **NXNoEffectFree()** as the freeing function (see **NXHashTablePrototype**).

NXPtrStructKeyPrototype and **NXStrStructKeyPrototype** identify prototypes that are useful for hash tables where the key is the first element of a structure and is either a pointer or a string.

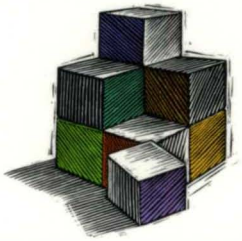
For example, **NXStrStructKeyPrototype** can be used to hash pointers to Example, where Example is:

```
typedef struct {
    char *key;
    int data1;
    ...
} Example
```

For **NXPtrStructKeyPrototype** and **NXStrStructKeyPrototype**, **NXReallyFree()** is used as the freeing function.

SEE ALSO **NXHashTablePrototype** and **NXCreateHashTable()**

NeXTSTEP Programming



NEXTSTEP GENERAL REFERENCE: RELEASE 3, VOLUME 1

NeXTSTEP is the object-oriented programming environment that speeds the development of all kinds of software—from mission-critical custom applications for business to advanced research projects for academia. NeXTSTEP offers building blocks that implement essential behavior in a variety of application areas—including database management, telecommunications and networking, and high-quality 2D and 3D graphics.

This first volume of the **NeXTSTEP General Reference** includes comprehensive descriptions of the application programming interface for the Application Kit and common classes. The second volume contains information on other kits, including the Database, Indexing, and 3D Graphics Kits.

The **NeXTSTEP Developer's Library** is essential reading for every NeXTSTEP enthusiast, providing authoritative, in-depth descriptions of the NeXTSTEP programming environment. Other titles in the **NeXTSTEP Developer's Library** include:

- **NeXTSTEP Development Tools and Techniques: Release 3**
- **NeXTSTEP User Interface Guidelines: Release 3**
- **NeXTSTEP Object-Oriented Programming and the Objective C Language: Release 3**
- **NeXTSTEP Operating System Software: Release 3**
- **NeXTSTEP Programming Interface Summary: Release 3**
- **NeXTSTEP Network and System Administration: Release 3**

NeXT develops and markets the industry-acclaimed NeXTSTEP object-oriented software for industry-standard computer architectures.

NEXTSTEP

Object Oriented Software

54495

9 780201 622201

ISBN 0-201-62220-3

US $44.95
CANADA $57.95

Addison-Wesley Publishing Company